Hardware-Aware AutoML for Efficient Deep Learning Applications

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Dimitrios Stamoulis

B.S., Electrical and Computer Engineering, National Technical University of Athens
M.Eng., Electrical and Computer Engineering, McGill University

Carnegie Mellon University
Pittsburgh, PA

May 2020

Acknowledgements

The past four years at Carnegie Mellon have been an incredible, transformative endeavor for me, and I could hardly enumerate the many ways in which these brilliant people have supported and inspired me. To these people I express my sincere gratitude.

First and foremost, I would like to thank my advisor Diana Marculescu for her wisdom and guidance. I cannot imagine a more supportive and inspiring advisor than Diana. Her trust and patience guided me in expanding my research interests and skills and allowed me to pursue novel research problems. I hope in my career to come close to the incredible role model she is as a mentor: always dedicated to helping her students to "unearth" their full potential. Thank you Diana for unveiling the true joy of a PhD to me.

I would like to thank my committee members, Gauri Joshi, Da-Cheng Juan, and Di Wang. I have greatly enjoyed my discussions with Gauri on the limitations and opportunities in the field of AutoML. Da-Cheng's research experience and insight greatly contributed to the early steps of this work. Through his enthusiasm, he has shown me how to energize team efforts and how to convey keen ideas and concerns in a constructive and sensitive manner. I was incredibly fortunate to work with Di, who taught me how simple intuition can spawn solutions to very challenging problems. His ability to identify and to motivate novel research directions allowed me to form ideas that eventually found their way into this thesis.

My deepest gratitude to Radu Marculescu, for his dedication to high-quality research and for continuously inspiring me to seek the big picture of the problems we worked on. As an EnyAC-er, I was fortunate to collaborate and interact with some incredibly talented researchers: Ermao Cai, Zhuo Chen, Ruizhou Ding, Ting-Wu Chin, Ahmet Fatih Inci, thank you for making our EnyAC meetings a feast of knowledge and for discussing and critiquing countless ideas with me through our various collaborations.

I have had some incredible mentors and collaborators during my internships: Florin Dartu, Rahul Krishnan, Jie Liu, and Dimitrios Lymberopoulos, thank you for giving me the opportunity to apply my research to real-world problems. I would also like to reiterate my sincere thanks to my past mentors, including Dimitrios Soudris, Dimitrios Rodopoulos, Zeljko Zilic, Brett Meyer, and Francky Catthoor, for being instrumental in fashioning my interests during my formative years as a young researcher.

I would also like to thank the following institutions and people for generously supporting my PhD: Qualcomm Inc. for the Qualcomm Innovation Fellowship 2018, the National Science Foundation (grants NSF CCF Grant 1314876, NSF CNS Grant No. 1564022, CSR Grant No. 1815780, and CCF Grant No. 1815899), the Pittsburgh Supercomputing Center (via NSF CCR Grant No. 180004P), David Barakat and LaVerne Owen-Barakat for the David Barakat and LaVerne Owen-Barakat Fellowship, and the Gerondelis Foundation. I would like to extend special thanks to Google Research for the Google Cloud Platform (GCP)

and TensorFlow Research Cloud (TFRC) credit awards. From the Google Pittsburgh office, many thanks to Chris Larkin and Jonathan Caton for their tremendous help. Big thanks to the awesome staff members in the ECE Department: Nathan Snizaski, Sherri Ferris, Judy Bandola, Bari Guzikowski, Shelley Phelps, and many others, for their help in administrative and student activities.

During my PhD, I was fortunate to meet numerous incredibly talented, genuine people that enriched my life both in Carnegie Mellon and in Pittsburgh. To Ifigeneia Apostolopoulou, Kartikeya Bhardwaj, Prashanth Mohan, Jeremie Kim, Vignesh Balaji, Sandeep Dsouza, Ching-Yi Lin, Ryan Kim, and many others, many thanks for the insightful discussions during classes and projects. To Mark Blanco, Vasu Agrawal, and other HKN members, thanks for the exciting activities with the CMU HKN chapter. To Amrit Pandey, Joe Sweeney, Antonis Manousis, Onur Kibar, Meric Isgenc, Pietro Simeoni, Luca Colombo, Martin Wagner, Tyler Vuong, Youngwook Do, Bechara Maroun, Marko Jereminov, Aayushya Agarwal, and many others, thank you for being the most amazing friends and for the amazing memories!

I find myself fortunate to have the guidance of Anastasios and Yannis Stamoulis, who have been instrumental in shaping my resilience, character, and ambitions. I want to reiterate a particular shout-out to my friends George Pavlakos, Vagelis Nikoloudakis, Yannis Chatzimichos, Nikos Tsiamitros, and Kostas Tokas, for the friendship we share since our first semester at NTUA and for their support throughout the peaks and valleys.

I am most thankful to my dear Katerina Mousteraki for her love and encouragement, for her patience and support through endless early mornings, submission deadlines, late nights, and countless miles covered across ever-changing internships, universities, and work locations around the globe. Thank you for being by my side throughout the whole journey.

Last, but certainly not least, my very special gratitude goes to my family for their unconditional love, guidance, and support: my father Anastasios Stamoulis, my mother Anastasia Aleiferi, and my brother Thomas Stamoulis, to whom I owe everything I have achieved in my life so far. This thesis is as much their hard work as it is mine.

Abstract

Deep Neural Networks (DNNs) have been traditionally designed by human experts in a painstaking and expensive process, dubbed by many researchers to be *more of an art than science*. However, the ever-increasing demand for state-of-the-art performance and real-world deployment has resulted in larger models, making the manual DNN design a daunting task. AutoML presents a promising path towards alleviating this engineering burden by automatically identifying the DNN hyperparameters, such as the number of layers or the type of layer-wise operations. As modern DNNs grow larger, AutoML methods face two *key challenges*: *first*, the increased DNN model sizes result in increased computational complexity during inference, making it difficult to deploy AutoML-designed DNNs to resource-constrained devices. *Second*, due to the large DNN design space, each AutoML search remains considerably costly, with an overall cost of hundreds of GPU-hours.

In this thesis, we propose AutoML methods that are both *hardware aware* and search-cost *efficient*. We introduce a Bayesian optimization (BO) methodology enhanced with hardware-cost predictive models, allowing the AutoML search to traverse the design space in a constraint "complying" manner, up to **3.5**$\times$ faster compared to vanilla BO methods. Moreover, we formulate the design of *adaptive* DNNs as an AutoML task and we jointly solve for the DNN architectures and the *adaptive* execution scheme, reducing energy consumption by up to **6**$\times$ compared to hand-tuned designs. Next, in a departure from existing *one-shot* Neural Architecture Search (NAS) assumptions on how the candidate DNN architectures are evaluated, we introduce a novel view of the *one-shot* NAS problem as finding the subsets of kernel weights across a *single-path one-shot* model. Our proposed formulation reduces the NAS search cost by up to **5,000**$\times$ compared to existing NAS methods. Taking advantage of such efficiency, we investigate how various design space and formulation choices affect the AutoML results, achieving a new state-of-the-art NAS performance for image classification accuracy (75.62%) under runtime constraints on mobile devices.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Deep Neural Networks (DNNs) have emerged as powerful models for numerous deep learning (DL) applications, such as image recognition [46], machine translation [126], object detection [73, 74, 45], and semantic segmentation [63]. The abundance of successful, real-world DNN-based products and applications in our daily lives (*e.g.*, 2D human pose estimation [55], Speech-to-Text APIs [39], real-time virtual-reality image rendering on head-mounted displays [81, 80, 88], to name a few) has ignited an ever-increasing interest in pushing the performance of DNNs to achieve state-of-the-art results. DNNs have been traditionally designed by human experts in an expensive and meticulous process, which has been dubbed by many researchers as *more of an art than science* [103]. However, as modern DNN models become increasingly deeper and larger, the task of hand-tailored DNN design has become a daunting challenge [119].

AutoML presents a promising path for alleviating the engineering costs and the complexity that are intrinsic to the design of neural networks, by automating the tuning of DNN hyperparameters (*e.g.*, the number of layers, the type of operations per layer, *etc.*) and by formulating the design of DNNs as a *hyperparameter optimization* problem [103, 97]. In fact, we are witnessing a proliferation of novel AutoML approaches, with formulations spanning many different methodologies, such as black-box hyperparameter optimization (HPO) [103, 106, 33, 62, 71, 5] and Neural Architecture Search (NAS) [138, 97, 79, 100]. Notably, commercial interest in AutoML has grown dramatically in recent years, as demonstrated by the vast computational resources committed to industry-driven AutoML research [138, 97, 1, 136, 131, 42, 35] and by the plethora of commercial cloud-based services and frameworks [32, 36, 86, 38, 121, 87]. Overall, AutoML methodologies constitute a research topic of paramount importance, since the commoditization of "push-button" DL solutions without the need for DL experts is expected to have significant reverberations across multiple industries.

1.1 Challenges for state-of-the-art AutoML frameworks

AutoML frameworks have currently established the state-of-the-art performance across numerous DL applications. Strong empirical results show that AutoML-generated DNNs outperform their hand-designed counterparts. However, despite the numerous beta tools available online [32, 86, 38], the growing demand for real-world deployment of DNNs has given rise to two key challenges in AutoML research.

Challenge 1 – Hardware efficiency of AutoML-designed DNNs: As modern DNNs become increasingly deeper, they require more computational power during inference. This increased computational cost becomes an impediment to the deployment of state-of-the-art AutoML-designed DNNs to resource-constrained devices, such as mobile phones and Internet-of-Things (IoT) nodes [122]. For instance, object classification can drain the smartphone battery within an hour [133]. Consequently, accounting for the hardware efficiency of AutoML-designed DNNs has emerged as an important research direction, as attested by the plethora of recent device-aware AutoML approaches [122, 55, 12, 31, 51].

In particular, existing methods revisit the earlier hardware-unaware AutoML formulations (*e.g.*, reinforcement learning [138] or evolutionary algorithms [97]) by incorporating the total FLOP count as a constraint and by optimizing for the so-called "mobile setting", *i.e.*, for models with less than 600 million parameters. Nonetheless, recent experimental results show that the number of operations or parameters does not approximate the latency well [31, 14, 8]. While recent methods explicitly account for hardware measurements and models (*e.g.*, inference latency) into AutoML formulations [31, 51, 128, 12], they have yet to address the second key challenge of prohibitively large search cost, as discussed next.

Challenge 2 – Efficiency in AutoML search cost: As state-of-the-art DNNs grow larger, their increased model complexity gives rise to a combinatorially large search space. For example, in the case of a mobile-efficient DNN with 22 layers, choosing among five candidate operations yields $5^{22} \approx 10^{15}$ possible DNN architectures [128]. Hence, AutoML methods need to tackle the intrinsically high cost of traversing this huge design space. Earlier methods use reinforcement learning (RL) to guide the exploration [122]. Nonetheless, training the RL controller poses prohibitive computational challenges, since tens of thousands of candidate DNNs need to be trained [128] for a total search cost of tens of thousands of GPU-hours. This immense amount of computational resources required for AutoML constitutes an unprecedented burden for cloud providers, with each search being as costly in terms of total carbon footprint as five round-trip flights from San Fransisco to New York [44].

Recent literature has seen a shift towards more sample-efficient differentiable formulations [79, 93, 132, 128, 12]. However, even the newest Neural Architecture Search (NAS) methods remain considerably expensive, with an overall cost of hundreds of GPU-hours [128, 12]. Consequently, existing AutoML

frameworks remain a bottleneck, especially when using them repeatedly to search for different hardware platforms under various device constraints: each single AutoML search [128, 12] has the same total carbon footprint as five cars during their lifetimes [44].

1.2 Thesis contributions

In light of the aforementioned challenges, in this thesis we strive to develop novel AutoML methods that are both *hardware aware* and search-cost *efficient*. We show that, by incorporating insight from the underlying search space, the hardware (HW) metrics, and the target applications, our proposed solutions outperform prior work in terms of DNN accuracy, while *significantly* reducing the search cost. We demonstrate this with the following novel **contributions**:

- **Chapter 3 – Hardware-enhanced Bayesian optimization**: We introduce a Bayesian optimization (BO) method where the HW-cost terms are directly incorporated into the BO formulation, allowing the AutoML search to traverse the design space in a constraint "complying" manner [109]. The proposed method reaches the near-optimal region **3.5**$\times$ faster compared to vanilla constrained BO methods.

- **Chapter 4 – AutoML for designing adaptive DNNs**: To the best of our knowledge, we are first to formulate the design of adaptive DNNs as an AutoML problem and to jointly solve for the architectures of the individual (heterogeneous) DNNs and the adaptive execution scheme under HW constraints [110]. We propose a BO-based methodology that reduces energy consumption by up to 6$\times$ and improves accuracy by up to 31.13% compared to hand-designed adaptive DNNs, when tested on a commercial NVIDIA mobile SoC and the CIFAR-10 dataset.

- **Chapter 5 – Single-path one-shot NAS**: We introduce a novel view of the *one-shot* NAS problem, drastically decreasing the number of trainable parameters. To the best of our knowledge, this is the *first* formulation of *one-shot* NAS as finding the subset of kernel weights in each DNN layer [113, 114]. The overall search cost (**8 epochs**) is up to **5,000**$\times$ **faster** compared to prior work.

- **Chapter 6 – State-of-the-art Mobile NAS performance**: We enhance the Mobile NAS search space by treating the Squeeze-and-Excitation [52] (SE) path as fully searchable. We show that larger SE ratios further improve the overall performance by yielding a better DNN accuracy-runtime trade-off and we investigate how various formulation choices affect the *one-shot* NAS performance [112]. Our method achieves a new state-of-the-art Mobile NAS accuracy on ImageNet classification (75.62%) with inference runtime on par with existing methods.

To this end, we believe that this thesis supports the following claim:

Thesis statement: AutoML methods can be both *hardware aware* and search-cost *efficient*.

1.3 Thesis organization

The remainder of this thesis is organized as follows.[1] Chapter 2 formulates the problem of hardware-constrained AutoML. Chapter 3 introduces our hardware-constrained BO methodology. Chapter 4 details the BO-based design of adaptive (heterogeneous) DNN systems under energy and communication constraints. In Chapter 5, we propose a novel *one-shot* NAS method. In Chapter 6, we detail an enhancement of the NAS design space that further improves the accuracy-runtime trade-off and we investigate how different implementation choices affect the performance of the AutoML-designed DNNs. Chapter 7 discusses the related work. Finally, Chapter 8 concludes this thesis and highlights directions for future research.

[1] This manuscript does not include some of my lead authored PhD research related to manufacturing process variations [115] and precipitation prediction models [29] as they fall outside the scope of this thesis.

Chapter 2

Background

AutoML methods treat the design of DNNs as a hyperparameter optimization problem. To facilitate the discussion in the following chapters, we first provide an overview of AutoML methods (Section 2.1), we then formulate the hardware-aware AutoML problem (Section 2.2), and we review some background material related to the key methods investigated in this thesis, *i.e.*, Bayesian optimization (Section 2.3) and *one-shot* NAS (Section 2.4).

2.1 Paradigms for AutoML

To contextualize the work in this thesis, we identify the main categories (and their respective sub-categories) in AutoML methods following the categorization by Hutter *et al*. [54]. We follow such dichotomies mainly to identify the focus for this thesis, and we note that different authors have drawn these delineations differently, *e.g.*, Li and Talwalkar identify the main AutoML components with respect to the search space, the search method, and the evaluation method of an AutoML approach [70]. As shown in Figure 2.1, AutoML can be categorized into three high-level categories, namely (i) black-box *hyperparameter optimization* (HPO), (ii) Neural Architecture Search (NAS), and (iii) *meta-learning*.

Hyperparameter optimization: HPO refers to the class of AutoML methods that tune a learning model over a **wide** range of hyperparameter choices related to *model architecture* (*e.g.*, number of parameters), *optimization* (*e.g.*, learning rate), and *regularization* (*e.g.*, regularization factors). HPO methods can be further separated based on their strategy to traverse the design space: *e.g.*, model-free [3], Bayesian optimization [34], and multi-fidelity [71], *etc*. Since every learning system has hyperparameters, HPO methods date back to the 1990s [65], spanning ML (*e.g.*, SVMs [106]) and DL applications (*e.g.*, DNNs [107]).

Neural Architecture Search: NAS methods refer to the recent sub-field of AutoML that aims to specifically tune the **structural** hyperparameters of DL models, *i.e.*, the *model architecture*. While there

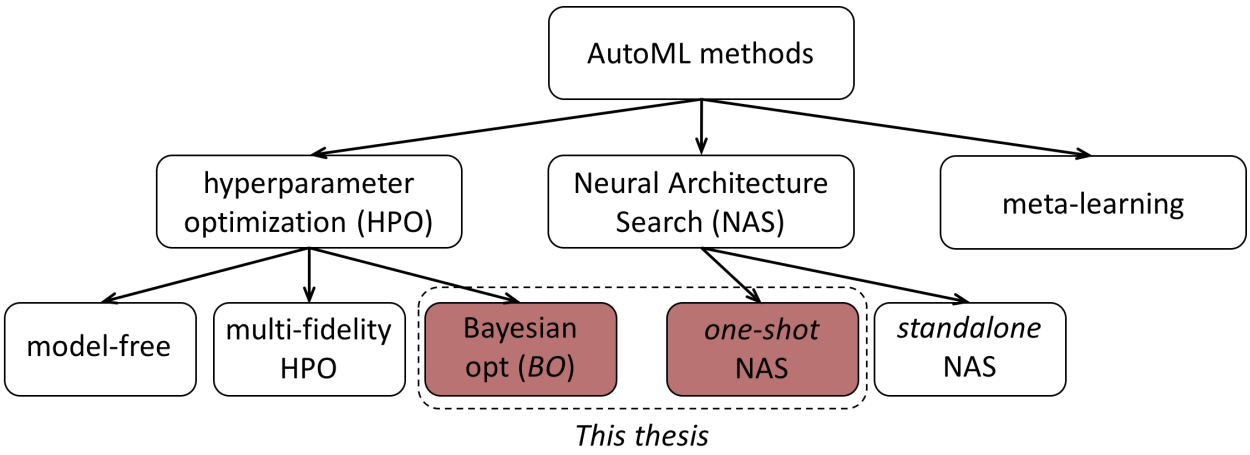

Figure 2.1: An overview of AutoML paradigms following the categorization by Hutter *et al*. [54]. In this thesis, we focus on AutoML based on Bayesian optimization (Chapters 3-4) and *one-shot* NAS (Chapters 5-6) techniques under hardware constraints.

is indeed a significant overlap between NAS and HPO methodologies, in this thesis we adhere to the following distinction between NAS and HPO from Hutter *et al*. [54]: in the former, the NAS search space consists of architectural hyperparameters **only**, while in the latter case (HPO) it does not.

As of this writing, NAS literature is witnessing a yearly exponential increase in the number of papers written on the subject [76]. Hence, we further categorize NAS works into *standalone* and *one-shot* NAS methodologies, based on how the candidate DNNs are evaluated. In particular, in *standalone* NAS, the search strategy (*e.g.*, reinforcement learning [137] or evolutionary algorithms [97]) selects a DNN in every iteration. The candidate design is trained from scratch on a (proxy) task to evaluate its performance, hence yielding computation demands on the order of thousands of GPU-days [138]. To reduce this computational burden, *one-shot* NAS treats all DNN architectures as subsets of a supernetwork (the one-shot model) [79]. This requires only the weights of a single one-shot model to be trained, and the various candidate designs are then evaluated by inheriting trained weights from the supernetwork.

Meta-learning: Meta-learning methods focus on *learning to learn* on a new task by building on experience obtained from related, previously studied tasks [125]. Several meta-learning strategies exhibit significant overlap with the previous two categories (*e.g.*, *warm-start* techniques in HPO [36] or *configuration-transfer* in proxy-based *one-shot* NAS [79]), hence in this thesis we consider only HPO and NAS.

Overall, this thesis exclusively focuses on BO-based HPO and *one-shot* NAS methodologies, as highlighted in Figure 2.1. Our choice stems from our goal to develop AutoML methods that are both *hardware-aware* and search-cost *efficient*. To this end, these two techniques are excellent starting points for our endeavor and the reason is twofold: (i) search-cost *efficiency* – prior work has shown that BO and *one-shot* NAS reduce

the search cost compared to other HPO and *standalone* NAS counterparts [79, 103]; (ii) hardware-awareness – both these methods have been previously used for AutoML under hardware constraints, which allows us to have representative comparisons with prior work by assessing our methodologies against well-studied AutoML search spaces and hardware platforms [92, 122]. In particular, we delve into BO (Chapters 3-4) and *one-shot* NAS (Chapters 5-6) techniques to design DNNs for image classification under hardware constraints.

2.2 Designing DNNs with hardware-aware AutoML: problem formulation

When designing DNNs, there is a plethora of options for the DL practitioner to choose from, such as:

- Layer-wise operation choice: convolution, pooling, fully-connected layer, *etc*.
- Types of convolution layers: varying number of filters, kernel size, expansion ratios, *etc*.
- Pooling layers: kernel size, stride values, *etc*.
- Fully-connected layers: number of units
- Learning hyperparameters: learning rate, momentum, weight decay, *etc*.

All these different tunable hyperparametes yield the design space $\mathcal{A}$ with all possible configurations.

In the context of automated hardware-constrained DNN design, our goal is to find the DNN architecture $a^* \in \mathcal{A}$ with the optimal learning performance with respect to the DL task (*e.g.*, the validation error on the target dataset) that satisfies the constraints (*e.g.*, power consumption during inference) on the target platform. Formally, we write the DNN design as a constrained optimization problem:

$$\min_{a \in \mathcal{A}} \ \mathcal{L}(a) \quad s.t. \quad \tilde{C}(a) \leq \tilde{C}_T \tag{2.1}$$

where our goal is to find the $a^* \in \mathcal{A}$ that minimizes the learning loss $\mathcal{L}$, and whose hardware-cost terms $\tilde{C}(a)$ satisfy the respective hardware constraints (target hardware performance) $\tilde{C}_T$. In practice, evaluating Equation 2.1 at different candidate designs a is costly. To this end, prior work has explored various methods to efficiently solve this problem.

2.3 Constrained Bayesian optimization

The key insight behind various black-box HPO methodologies[103, 106, 5] is to approximate the objective and constraint terms by surrogate probabilistic models which are cheaper to evaluate. In particular, *Bayesian optimization* (BO) is a sequential model-based approach that has been shown to work well in practice for problems with few tens of hyperparameters [103].

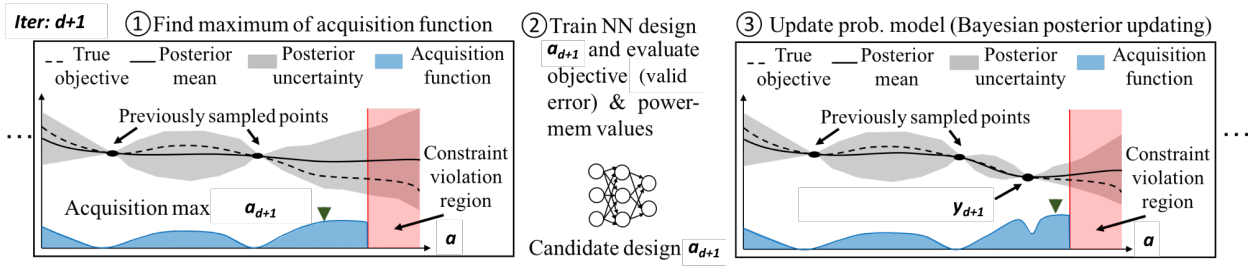

Figure 2.2: Overview of BO during each iteration $d+1$. BO uses a surrogate probabilistic model $\mathcal{M}_i$ to approximate each term; the plots show the mean and confidence intervals estimated with the model (the true objective function is shown for reference, but it is unknown in practice). At each iteration $d+1$, an acquisition function $q(\cdot)$ is expressed based on the model $\mathcal{M}_i$ and the maximizer of $q(\cdot)$ is selected as the candidate design point a_{d+1} to evaluate. The objective is evaluated, *i.e.*, the candidate DNN a_{d+1} is trained and evaluated on the validation set. Then, the probabilistic models $\mathcal{M}_i$ are refined via Bayesian posterior updating based on the new observation. The process is repeated for $D_{\max}$ steps.

To solve the constrained optimization problem in Equation 2.1, BO approximates each objective and constraint term with a probabilistic (surrogate) model $\mathcal{M}_i$ based on Gaussian processes (GP). For each function term f_i, the GP model is a probability distribution over the possible functions of $f_i(a)$, and it approximates each f_i at each iteration $d+1$ based on data $\mathbf{A} := a_j \in \mathcal{A}_{j=1}^{d}$ queried so far. We assume that the values $\mathbf{f}_i := f_{i,\{1:d\}}$ of a function term f_i at points $\mathbf{A}$ are jointly Gaussian with mean $\mathbf{m}_i$ and covariance $\mathbf{K}_i$, *i.e.*, $\mathbf{f}_i \mid \mathbf{A} \sim \mathcal{N}(\mathbf{m}_i, \mathbf{K}_i)$. This formulation intuitively encapsulates our belief about the shape of functions that are more likely to fit the data observed so far. Since the observations $\mathbf{f}_i$ are noisy with additive noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, we write each GP model as $\mathbf{y}_i \mid \mathbf{f}_i, \sigma^2 \sim \mathcal{N}(\mathbf{f}_i, \sigma^2\mathbf{I})$. At each point a_j, GP gives us a cheap approximation for the mean and the uncertainty of the respective term, written as $p_{\mathcal{M}_i}(y_i|a_j)$ and illustrated in Figure 2.2 with the black curve and the grey shaded areas.

Each iteration $d+1$ of a BO algorithm consists of three key steps:

1. **Maximization of acquisition function**: We first need to select the point a_{d+1} (*e.g.*, the next candidate DNN configuration) at which the objective (*e.g.*, the validation error of the candidate DNN) will be evaluated next. This task of guiding the search relies on the so-called acquisition function $q(a)$. A popular choice for the acquisition function is the Expectation Improvement (EI) criterion, which computes the probability that a term f_i will exceed (negatively) some threshold y_i^+, *i.e.*, $EI(a) = \int_{-\infty}^{\infty} \max\{y_i^+ - y_i, 0\} \cdot p_{\mathcal{M}_i}(y_i|a)\, dy_i$. Intuitively, $q(a)$ provides a measure of the direction toward which there is an expectation of improvement of the objective function. The acquisition function is evaluated at different candidate points a, yielding high values at points where the GP's uncertainty is high (*i.e.*, favoring exploration), and where the GP predicts a high objective (*i.e.*, favoring exploitation) [103]; this is qualitatively illustrated in Figure 2.2 (blue curve). We select the maximizer of $q(a)$ as the point a_{d+1} to evaluate next (green triangle).

2. **Evaluation of the objective/constraints**: Once the current candidate DNN architecture a_{n+1} has been

selected, the DNN is generated and trained to completion to evaluate its objective and constraint terms. Please note that this step is the main bottleneck in AutoML. Hence, the efficiency of our proposed BO-based methods (Chapters 3-4) comes from detecting when this step can be bypassed or how the search can be quickly guided towards constraint-satisfying samples (*i.e.*, fewer function evaluations).

3. **Probabilistic model update**: As the new term value $y_{i,\{d+1\}}$ becomes available at the end of iteration $d+1$, the probabilistic model $p_{\mathcal{M}_i}(y_i)$ is refined via Bayesian posterior updating (the posterior mean $\mathbf{m}_{i,\{d+1\}}$ and covariance $\mathbf{K}_{i,\{d+1\}}$ can be analytically derived). This step is quantitatively illustrated in Figure 2.2 with the black curve and the grey shaded areas. Please note how the updated model has reduced uncertainty around the previous samples and newly observed point. For a comprehensive discussion of GP models the reader is referred to [95].

Overall, the above methodology has been successfully used in several HPO tasks and it is a key component in various commercial or industry-supported tools [32, 87], hence we delve into hardware-constrained BO problems in this thesis.

2.4 *One-shot* Neural Architecture Search (NAS)

To improve the search cost of AutoML methods, *one-shot* NAS exploits *weight-sharing* by relaxing the categorical DNN design choices to a one-shot differentiable optimization problem. Specifically, one-shot methods relax the combinatorial optimization problem (categorical hyperparameters a) to a softmax-based approximation parameterized by α, *i.e.*, each hyperparameter is $a_k \approx \text{softmax}(\alpha_k)$. Next, an over-parameterized *multi-path* supernet is constructed (illustrated in Figure 2.3) where, for each layer, every candidate operation is added as a *separate* trainable path [12, 79, 128]. Hence, in these methods, which we refer to differentiable *one-shot* NAS [79, 93], the DNN design becomes an operation/path selection problem.

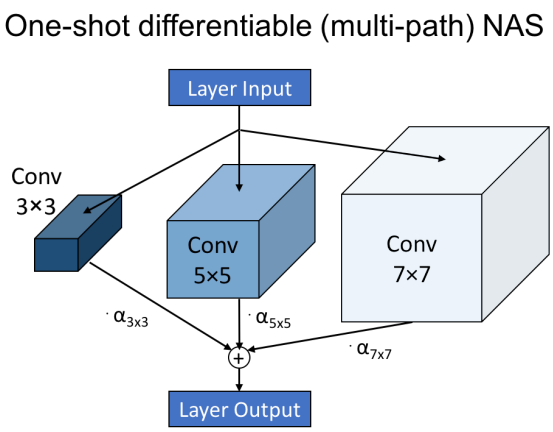

Figure 2.3: One-shot differentiable NAS relaxes the combinatorial optimization problem (Equation 2.1) to a softmax-based approximation of the DNN choices. A supernet model is constructed that encompasses all candidate DNN designs, *i.e.*, all solutions $a \in \mathcal{A}$.

For layer input $\mathbf{x}$, the output of each layer i is a (weighted) sum defined over the output of N different paths, where each path j corresponds to a different candidate kernel $\mathbf{w}^{i,j}$. The weight of each path $\alpha^{i,j}$

corresponds to the probability that this path is selected over the parallel paths:

$$o^{i}_{multi-path}(\mathbf{x}) = \sum_{j=1}^{N} \alpha^{i,j} \cdot o^{i,j}(\mathbf{x}) = \alpha^{i,0} \cdot \text{conv}(\mathbf{x}, \mathbf{w}^{i,0}_{3\times3}) + \cdots + \alpha^{i,N} \cdot \text{conv}(\mathbf{x}, \mathbf{w}^{i,N}_{5\times5}) \tag{2.2}$$

Problem formulation: NAS formulations solve for the (distributions of) paths of the *multi-path* supernet that yield the optimal architecture, *i.e.*, for the optimal architecture parameters α (path weights), such that the weights w_α of the corresponding α-architecture have minimal loss $\mathcal{L}(\alpha, w_\alpha)$. However, solving for a constrained case (*e.g.*, Equation 2.1) yields DNNs specific to the given constraints $\tilde{C}_T$, and the hyperparameter optimization process has to be repeated for a different $\tilde{C}_T$ values. To this end, given the computational cost of performing architecture search, NAS literature has been more interested in finding multiple Pareto-optimal solutions in a single search [122]. Hence, NAS methods revisit the constrained Equation 2.1 and they solve instead for a weighted (trade-off) objective:

$$\min_{\alpha} \min_{w_\alpha} \mathcal{L}(\alpha, w_\alpha) + \lambda \cdot \tilde{C}(a) \tag{2.3}$$

Solving Equation 2.3 gives rise to a challenging *bi-level* optimization problem [79]. Existing methods interchangeably update the α's while freezing the w's and vice versa, leading to several gradient steps. As expected, branching out all paths is intrinsically inefficient, since the number of trainable parameters that need to be maintained and updated during the search grows linearly with respect to the number of candidate operations per layer [1], leading to memory explosion [12] and increased search cost, with an overall computational demand of hundreds of GPU-hours. In Chapters 5-6 we introduce our novel NAS approach to alleviate the search cost related to *one-shot* differentiable NAS approaches.

Mobile NAS design spaces

Typically, NAS methods start from a fixed DNN "backbone" and they choose each layer-wise operation from a set of predefined, candidate operations. A typical "backbone" choice for hardware-aware NAS is the MobileNetV2 design [102] (Figure 5.4).[1] **MobileNetV2 skeleton**: In this macro-architecture, except for the head and stem layers, all DNN layers are grouped into blocks based on their input resolutions and the filter sizes. The filter numbers per block follow the values in [128], *i.e.*, we use seven blocks with up to four layers each. Each blocks consists of four mobile inverted bottleneck convolution MBConv [102] layers. The MBConv micro-architecture consists of a point-wise (1×1) convolution, a $k \times k$ depthwise convolution, a Squeeze-and-Excitation (SE) block [52], and a linear 1×1 convolution. Unless the layer has a stride value of two, a skip path is introduced to provide a residual connection from input to output.

[1] Different types of search spaces and DNN "backbones" have been previously considered (*e.g.*, hierarchical [78], cell-based [138], *etc.*). However, merging intermediate results across multiple branches of the cell/hierarchy with addition and concatenation operations has been shown to be hardware costly [128]. Thus, such NAS search spaces are beyond the scope of our work.

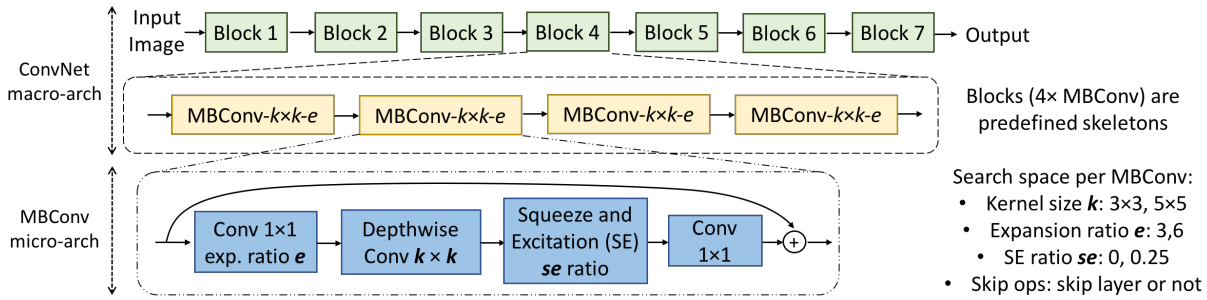

Figure 2.4: **MobileNetV2-based search space** [102]: DNN layers are grouped into predefined blocks, based on their filter sizes. Each block contains four mobile inverted bottleneck convolution **MBConv** layers. For each MBConv, NAS methods search for the convolution kernel size k, the expansion ratio e, whether the layer is skipped or not, *etc*. MBConv from different blocks/layers can be different.

In Figure 2.4 (right) we list the various layer-wise design choices. Each MBConv is parameterized by: (i) the kernel size of the depthwise convolution $k \times k$, (ii) the expansion ratio e, *i.e.*, the ratio between the output and input of the first 1×1 convolution, and (iii) the Squeeze-and-Excitation [52] ratio se, *i.e.*, the ratio between the number of channels in the intermediate convolution and the input of the Squeeze-and-Excitation path. NAS also considers a special *skip-op* "layer", which "zeroes-out" the kernel and feeds the input directly to the output, *i.e.*, the entire layer is dropped. This choice effectively corresponds to reducing the depth of the network. Based on this parameterization, we denote each MBConv as MBConv-$k \times k$-e-se. The goal of NAS is to automatically identify the type of each MBConv layer in the DNN design.

Chapter 3

Hardware-Constrained DNN Hyperparameter Optimization via Bayesian Optimization

Bayesian optimization (BO) has been the prominent hyperparameter optimization (HPO) method for *model selection*. However, vanilla BO relies on costly function evaluations to learn the probability that a candidate Deep Neural Network (DNN) architecture satisfies the hardware constraints. In this Chapter, we introduce a novel BO formulation that explicitly incorporates hardware models, allowing the hyperparameter solver to efficiently navigate the design space in a hardware-constraint "complying" manner.

3.1 Chapter overview

Bayesian optimization (BO) refers to a class of black-box HPO methods where the objective and constraint functions are only accessible via expensive point evaluations. Ever since the earlier Machine Learning (ML) applications posed the challenge of selecting the hyperparameters of SVMs or regression models, the quintessential application for BO has been *model selection* [106]. In particular, BO naturally fits hyperparameter tuning problems, where it is challenging to analytically capture the generalization performance of an ML model under various Euclidean or categorical hyperparameter choices.

Following the surge of interest in deep learning (DL), Deep Neural Networks (DNNs) have emerged as powerful models in supervised learning applications, such as image classification [46] and object detection [73, 74, 45]. Hence, in the context of DNN design, this motivates viewing *model selection* as an HPO problem, *i.e.*, studying *model selection* over the space of DNN architectures to optimize for generalization performance. Naturally, BO has been one of the earlier methods to successfully outperform human experts [119], as attested by earlier BO-based AutoML tools (*e.g.*, HyperOpt [4], Spearmint [106], Google Vizier [36]).

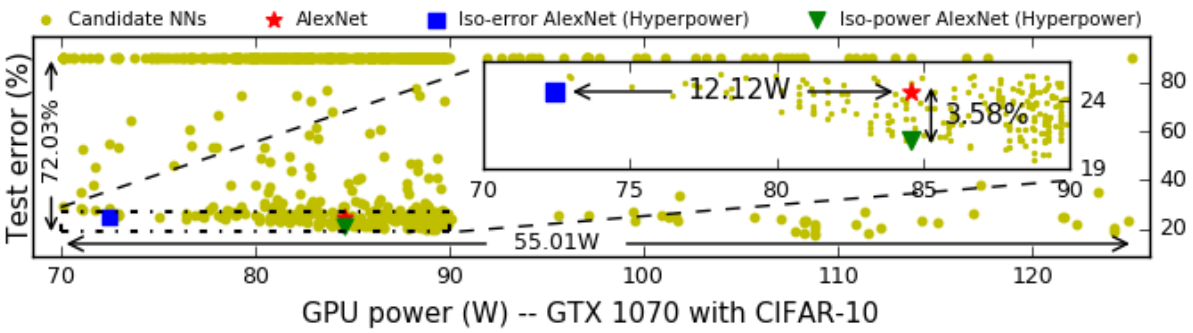

Figure 3.1: Visualizing the complexity of the design space – testing error and GPU power consumption for different DNN design based on the AlexNet search space (CIFAR-10 with Caffe [56] on NVIDIA GTX-1070).

In the context of hardware-constrained BO, existing methods treat the hardware constraints as *low-cost*, *a priori* known constraints, whose probability that are satisfied is modeled during the search by inexpensive point evaluations. This is based on the observation that profiling the hardware cost of a DNN architecture is considerably cheaper than training the network to evaluate its accuracy. Existing methodologies exploit this intuition by profiling DNNs on simple datasets such as MNIST or by using the DNN parameter count as a *low-cost* proxy of the hardware efficiency [48, 96]. Nonetheless, as shown in Figure 3.1, the design space could be more complex when considering the DNN power consumption on an actual GPU platform and for a larger dataset. For instance, we observe that, for a given accuracy level, the DNN power consumption could differ significantly by up to 55.01W (*i.e.*, more than a third of the GPU Thermal Design Power). As discussed in our results later in this Chapter, the low cost of hardware performance evaluations could become non-trivial to exploit under power or memory consumption constraints.

3.1.1 Key novelty: Enhancing BO with hardware models

We conjecture that *the low evaluation cost of hardware metrics can be further exploited to directly train predictive models of the power and memory consumption as a function of the DNN hyperparameters*. These models can be in turn incorporated directly into a novel BO formulation, hence allowing the BO solver to efficiently navigate the design space in a constraint "complying" manner. This gives rise to two interesting challenges: *first*, how to accurately model the power and memory consumption of DNNs (during inference) based on profiling data from commercial GPU platforms. *Second*, how to explicitly incorporate the predictive hardware-cost models into the BO formulation without introducing additional cost that could hamper the efficiency of the BO solver. To answer these questions, we develop a novel BO framework, namely *HyperPower* [109]. In *HyperPower*, we propose accurate regression-based predictive models and a novel constraint-aware acquisition function.

3.1.2 Contributions and Chapter organization

To the best of our knowledge, this thesis makes the following contributions:

1. We introduce predictive models to capture the power and memory consumption of DNNs running on GPUs. Our regression-based models are accurate across various GPU platforms and image classification datasets, with an overall prediction error less than 7%.

2. We propose a hardware-aware BO formulation that efficiently traverses the design space in a constraint "complying" manner. *HyperPower* reaches the near-optimal region up to **3.5**$\times$ faster compared to vanilla constrained BO methods.

The organization of this Chapter is as follows: Section 3.2 formulates the problem and investigates the DNN power consumption as a *low-cost* constraint. Section 3.3 introduces *HyperPower*, including the regression-based predictive models and the constraint-aware acquisition function. Section 3.4 demonstrates the experimental results. Last, Section 3.5 provides the discussion.

3.2 Hardware-constrained Bayesian optimization

The power $P(\cdot)$ and the memory $M(\cdot)$ consumption of a DNN (during inference) have been traditionally seen as key impediments to deploying DNNs to low-power devices, such as IoT nodes. Hence, a common AutoML use case is to solve Equation 2.1 under power and memory constraints, P_T and M_T, respectively:

$$\min_{a \in \mathcal{A}} \ \mathcal{L}(a) \quad s.t. \quad P(a) \leq P_T, M(a) \leq M_T \tag{3.1}$$

Low-cost constraint evaluation in vanilla Bayesian optimization baseline

Power as low-cost constraint: We first investigate whether the power consumption can be formulated as an *a priori* known constraint. By randomly sampling and training candidate designs a on the MNIST dataset [68], we measure how the (inference) DNN power and the validation accuracy vary throughout training. As shown in Figure 3.2, we observe that the DNN power consumption measured on an NVIDIA TX1 device does not significantly

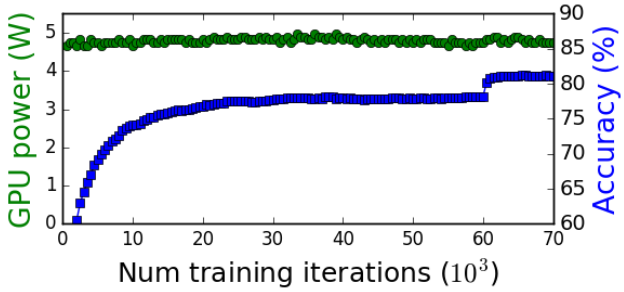

Figure 3.2: DNN power consumption as a *low-cost* constraint (power measured on NVIDIA TX1).

change as the DNN is trained for more iterations. That is, the power characteristics of a DNN are not affected by the quality of the trained model itself.

This observation allows us to formulate the power-constrained optimization (and also memory-constrained, as discussed in our results) as BO with *a priori* known constraints. During the search, the low-cost constraints are fitted with GPs [34], using a latent function for each constraint. Specifically, each GP captures the probability of the constraint being satisfied, *i.e.*, $Pr(M \leq M_T)$ and $Pr(P \leq P_T)$. In each iteration, we first evaluate the low cost constraints of power and memory consumption. We refer to this baseline as *vanilla* constrained BO [34]. This method has been previously used for FLOP-constrained DNN design [34, 48], since similarly the FLOP count is a low-cost constraint.

3.3 Proposed methodology: *HyperPower*

To enhance BO with "explicit hardware awareness", we first propose to model power $P(\cdot)$ and and the memory $M(\cdot)$ consumption of a DNN (during inference) as a function of the DNN hyperparameters. Specifically, we model $P(\cdot)$ and $M(\cdot)$ as a function of the J discrete (structural) hyperparameters $\hat{a} \in \mathbb{Z}_+^J$ (subset of $a \in \mathcal{A}$); we train on the structural hyperparameters $\hat{a}$ that affect the DNN power and memory (*e.g.*, number of hidden units), since parameters such as learning rate have negligible impact. To this end, we employ offline random sampling by generating different configurations in the designs space. Per candidate design $\hat{a}_l$, we measure the power P_l and memory M_l consumption values during inference on different GPU platforms. Given the L profiled data points $\{(\hat{a}_l, P_l, M_l)\}_{l=1}^L$, we train linear regression models parameterized by parameters $\mathbf{p}, \mathbf{m} \in \mathbb{R}^J$, *i.e.*:

$$\textbf{Power model}: \quad \mathcal{P}(\hat{a}) = \sum_{j=1}^{J} p_j \cdot \hat{a}_j \tag{3.2}$$

$$\textbf{Memory model}: \quad \mathcal{M}(\hat{a}) = \sum_{j=1}^{J} m_j \cdot \hat{a}_j \tag{3.3}$$

We train the models (Equations 3.2-3.3) with a 10-fold cross validation on the dataset $\{(\hat{a}_l, P_l, M_l)\}_{l=1}^L$. While we have experimented with nonlinear regression [8], the linear models provide sufficient accuracy (as shown in our results). More importantly, the linear formulation is cheap to evaluate with the acquisition function on different design points $a \in \mathcal{A}$ (as discussed in subsection 3.3.2).

3.3.1 Proposed power and memory models

To obtain datapoints we profile the power and memory consumption of various DNN designs on two different machines, *i.e.*, a server machine with an NVIDIA GTX 1070 and a low-power embedded board

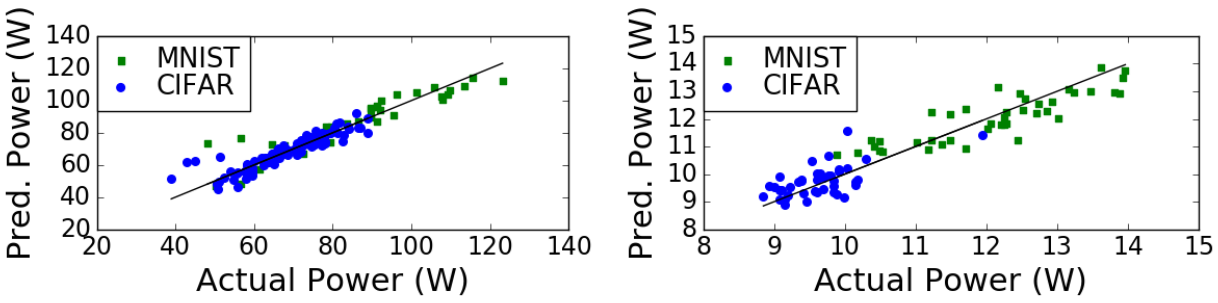

Figure 3.3: Actual and predicted power using our models for MNIST and CIFAR-10, executing on GTX 1070 (left) and Tegra TX1 (right).

Table 3.1: Root Mean Square Percentage Error (RMSPE) values of the proposed power and memory models.

Model	MNIST GTX 1070	CIFAR-10 GTX 1070	MNIST Tegra TX1	CIFAR-10 Tegra TX1
Power	5.70%	5.98%	6.62%	4.17%
Memory	4.43%	4.67%	– –	– – [1]

NVIDIA Tegra TX1. We profile the DNNs offline using `Caffe` [56] on both the CIFAR-10 and MNIST. To capture points of the design space, we profile variants of the AlexNet network for MNIST and CIFAR-10, by randomly varying the size of the different layers (details in the experimental results section).

In Figure 3.3, we plot the predicted and actual power values, trained on the MNIST and CIFAR-10 networks for both GTX 1070 and Tegra TX1.[1,2] Alignment across the blue line indicates good prediction results. We observe good prediction for all tested platforms and datasets, with a Root Mean Square Percentage Error (RMSPE) value always less than 7% (Table 3.1) for both power and memory models. It is worth noticing the power value ranges per device and that our proposed models can accurately capture both the high-performance and low-power design regimes.

3.3.2 Proposed constraint-aware acquisition function

Inspired by constraint-aware heuristics [34, 40], we propose a power and memory constraint-aware acquisition function:

$$q(a) = \int_{-\infty}^{\infty} \max\{y^+ - y, 0\} \cdot p_{\mathcal{M}}(y|a) \cdot \mathbb{I}[\mathcal{P}(\hat{a}) \leq P_T] \cdot \mathbb{I}[\mathcal{M}(\hat{a}) \leq M_T] \, dy \tag{3.4}$$

where $\hat{a}$ are the structural hyperparameters, $p_{\mathcal{M}}(y|a)$ is the predictive marginal density of the objective function at a based on surrogate model M. $\mathbb{I}[\mathcal{P}(\hat{a}) \leq P_T]$ and $\mathbb{I}[\mathcal{M}(\hat{a}) \leq M_T]$ are indicator functions, which are equal to 1 if the power and memory constraints are respectively satisfied. The threshold y^+

[1]Tegra does not support NVML API for memory measurements, and the `tegrastats` command reports utilization and not memory consumption.

[2]Due to low GPU utilization for the MNIST models, we profile candidate DNNs on MNIST with an inference batch size of 4. The DNNs on CIFAR-10 are profiled with batch size 1.

is adaptively set to the best value $y^+ = \max_{j=1:d} y_j$ over previous observations [103][34]. Intuitively, we capture the fact that improvement should not be possible in regions where the constraints are violated.

3.3.3 Early termination enhancement

Last, we observe that candidate architectures that diverge during training can be *quickly identified* only after a few training epochs (Figure 3.4). Please note that this is different than predicting the final test error of a network, which could suffer from overestimation issues [30], introducing artifacts to the probabilistic model. Instead of predicting for converging cases, we identify diverging cases, allowing the optimization process to discard low-performance samples. We incorporate early termination into *HyperPower* to further enhance the efficiency of our approach.

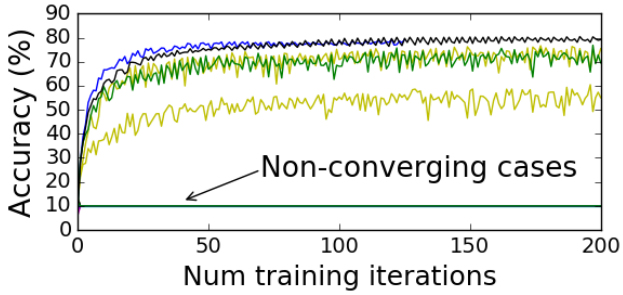

Figure 3.4: Early termination insight: how accuracy can indicate configurations that do not converge to high-accuracy values ($> 10\%$).

3.4 Experimental results

3.4.1 Experimental setup

For a detailed evaluation of our proposed methodology, we compare *HyperPower* against the vanilla BO baseline (Subsection 3.2). We also consider two random search variants. First, we consider constrained *Random Search* with early termination of constraint-violating samples. Moreover, we consider *Random Walk*, a random-based method that aims to "tame" the randomness by tuning the exploitation-exploration trade-off [105]; each next random point a_{d+1} is selected around the point a^+ with the best objective value y^+ over previous observations. Formally, at any step we select from within "neighborhood" controlled by σ_0^2, *i.e.*, $a_{d+1} \sim \mathcal{N}(a^+, \sigma_0^2)$.

We implement these methods on top of `Spearmint` [106]. We implement wrapper scripts around the objective/constraint functions that are queried by `Spearmint`, that automate the generation of `Caffe` simulations, and power/memory model evaluations. We employ hyperparameter optimization on variants of the AlexNet network for MNIST and CIFAR-10, with six and thirteen hyperparameters respectively. For the convolution layers we vary the number of features (20-80) and the kernel size (2-5), for the pooling

layers the kernel size (1-3), and for the fully connected layers the number of units (200-700). We also vary the learning rate (0.001-0.1), the momentum (0.8-0.95), and the weight decay (0.0001-0.01) values.

3.4.2 *HyperPower* outperforms vanilla Bayesian optimization & random search

Fixed number of function evaluations: We first assess the effectiveness of *HyperPower* compared to vanilla BO and random search under a fixed number of function evaluations. We optimize for the validation error on CIFAR-10 with a power constraint of 90W. As typical values for the experiments [34, 106, 30], we select a maximum number of 50 iterations per run. We plot the progress of each method in Figure 3.5, where we observe that the proposed methodology reaches the near-optimal region faster compared to both vanilla BO and random search methods. To understand such efficiency of *HyperPower*, we visualize the constrained violating samples per method in Figure 3.6. In particular, we confirm in Figure 3.6 (left) that *HyperPower* does not select samples that violate the constraint. In Figure 3.6 (right), we plot the validation error and the power consumption of each DNN considered. Once again, we see that *HyperPower* queries design points (red circles) to the left of the power constraint (90*W*). Last, as expected, we note that the random search methods are more likely to sample suboptimal points compared to the BO ones. Overall, the proposed formulation (Equation 3.4) benefits the efficiency of the BO solver, since it allows *HyperPower* to reach the average best error in a **fifth** of function evaluations (Figure 3.5).

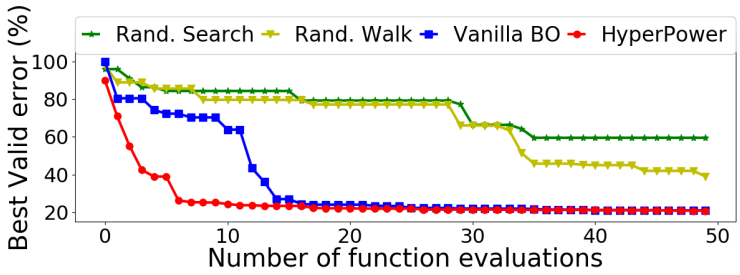

Figure 3.5: *HyperPower* reaches the near-optimal region in a fifth of point evaluations compared to vanilla BO and random search.

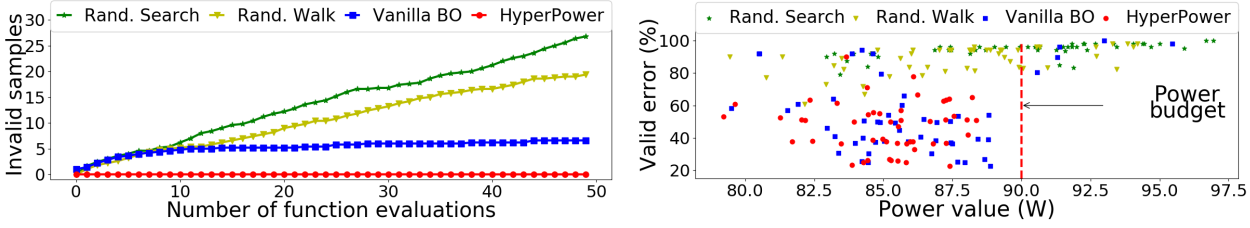

Figure 3.6: Assessment of *HyperPower* compared to random search and vanilla BO under fixed number of point evaluations on CIFAR-10 CNN: (left) Number of constraint-violating samples against the number of point evaluations. (right) Validation error and power value per candidate design.

Fixed wall-clock runtime budget: We evaluate the different methods under maximum wall-clock runtime budget, *i.e.*, two and five hours for MNIST and CIFAR-10, respectively. This scenario is important

in a more commercial-standard context when executing on a cluster [103] and under pricing schemes in cloud systems [119, 30]. We repeat the exploration for three runs per method for each considered device-dataset pair with the following constraints constraints: 85W and 1.15 for MNIST on GTX 1070, 90W and 1.25GB for CIFAR-10 on GTX 1070, 10W for MNIST on Tegra TX1, and 12W for CIFAR-10 on Tegra TX1 (no memory constraints on Tegra). In Table 3.2 we report the mean and the standard deviation of the best test error achieved by each method. We observe that *HyperPower* achieves the best mean test-error. We can also note that our method has less variance in all but one case. For the random methods, this is because some runs failed to find a high-performance region.

Table 3.2: Mean best test error (and standard deviation in parenthesis) achieved per method.

Solver	MNIST – GTX 1070	CIFAR-10 – GTX 1070	MNIST – Tegra TX1	CIFAR-10 – Tegra TX1
Random search [3]	1.01% (0.18%)	24.39% (3.08%)	0.97% (0.14%)	24.09% (1.97%)
Random Walk [105]	0.84% (0.08%)	22.88% (0.87%)	0.90% (0.12%)	21.90% (0.59%)
Vanilla constrained BO [106]	0.85% (0.12%)	22.09% (0.35%)	0.91% (0.07%)	22.99% (0.41%)
HyperPower (proposed)	**0.81%** (0.02%)	**21.81%** (0.38%)	**0.79%** (0.03%)	**21.95%** (0.65%)

Finally, we evaluate the benefit of combining early termination with the predictive models in the our BO formulation. In Figure 3.7, we plot the best validation error (%) with respect to the total hyperparameter optimization time for CIFAR-10 on GTX 1070 (5-hour execution) for both the *vanilla* BO, *i.e.*, BO without these two enhancements (predictive models and early termination), and *HyperPower*. We observe that our proposed methodology reaches low-error samples faster that the default (exhaustive) BO. We note the density of the samples at the beginning of the red line; this is to be expected, since low-performance or violating samples can be quickly discarded thanks to the introduced enhancements.[3] Overall, our proposed methodology reaches the near-optimal region **3.5**× faster than the *vanilla* method.

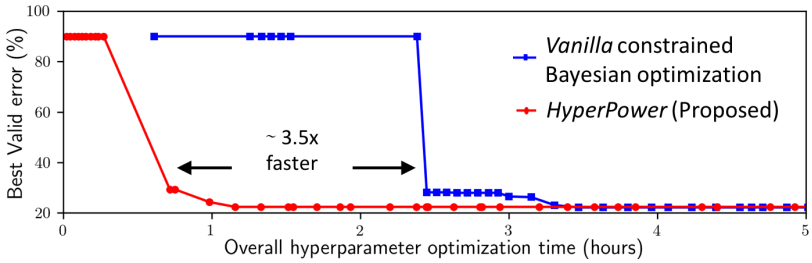

Figure 3.7: *HyperPower* reaches low-error samples faster compared to *vanilla* constrained BO, capturing the benefit of using early termination and predictive hardware cost models.

[3] We note that early termination can individually improve the baselines methods, *i.e.*, BO without predictive models and the random search methods. The reader is referred to [109] for detailed ablation studies.

3.5 Discussion

In this Chapter, we introduce *HyperPower*, a framework that enables efficient hardware-constrained BO for DNN design. We show that power consumption can be used as a low-cost, *a priori* known constraint, and we proposed predictive models for the power and memory of DNNs executing on GPUs. By incorporating the predictive models into a hardware-constrained BO formulation, *HyperPower* reaches the near-optimal region **3.5**$\times$ faster than the *vanilla* method. One straightforward direction for future work is to adapt our proposed BO framework to a more complex design space. In Chapter 4, we investigate BO in a novel image classification paradigm based on *adaptive* DNNs.

Chapter 4

Hardware-Constrained Adaptive DNNs Design

Adaptive Deep Neural Networks (DNNs) have been recently introduced as means for hardware-efficient image classification. Adaptive DNNs refer to systems of DNNs with different accuracy and computation characteristics, where a selection scheme *adaptively* selects the network to be evaluated for each input image. In this Chapter, we pursue a more powerful design paradigm where both the DNN architectures and the selection scheme are jointly optimized. To solve this problem, we adapt our BO-based formulation to efficiently design adaptive DNNs under energy, accuracy, and communication constraints.

4.1 Chapter overview

The energy requirements of DNNs have emerged as a key impediment preventing their deployment on energy-constrained embedded and mobile devices, such as Internet-of-Things (IoT) nodes, wearables, and smartphones. As means to reducing the average classification cost, prior art has observed that a significant portion of an image classification dataset can be correctly classified by simpler (computationally efficient) DNNs, while **only** a fraction of examples require larger (computationally expensive) DNNs [37]. Such intuition has motivated the introduction of adaptive DNNs, *i.e.*, systems of DNNs with different accuracy and computation characteristics, where a selection scheme is responsible for choosing which network classifies each input image [7]. Adaptive DNNs have been featured in numerous image classification frameworks from both industry [37] and academia [127, 90, 89, 92, 91].

Nevertheless, all previous approaches focus on learning how data should be processed among the DNNs, hence only optimizing with respect to the selection scheme while treating each DNN as a *blackbox* (*i.e.*, pre-trained off-the-shelf DNN). This could severely impede the effectiveness of adaptive DNNs when used in real-world applications. In fact, our analysis in this Chapter shows that when deployed on energy-constrained mobile devices, *adaptive* execution allows for only a small headroom for energy consumption

savings [110]. For example, our results show that existing methods, if tested on commercial NVIDIA Tegra TX1 platforms, could actually lead to an increase in energy consumption under strict accuracy constraints.

4.1.1 Key novelty: adaptive DNNs as a hyperparameter optimization problem

In a departure from existing assumptions on how to design adaptive DNNs, we make the following **novel** observation: *the hardware efficiency of adaptive DNNs can be further improved if the DNN architectures are optimized jointly with the network selection scheme*. This insight gives rise to a challenging question: "can we formulate and efficiently solve the design of adaptive DNNs as a hyperparameter optimization problem?" In the forthcoming sections of this Chapter we will successfully answer this question, by developing a novel BO framework to efficiently design adaptive DNNs for edge-computing nodes [110].

4.1.2 Contributions and Chapter organization

In this thesis, we make the following contributions:

1. To the best of our knowledge, we are *first* to jointly optimize the DNN architectures and the selection scheme in adaptive DNNs, by formulating the process as an AutoML problem. We show that our formulation is generic and able to incorporate different design goals: in our evaluation, we consider optimization under energy, accuracy, and communication constraints for image classification on edge-computing nodes.

2. We propose a hardware-aware BO method to exploit properties of the mobile space: we observe that once the accuracy and hardware measurements have been obtained for a set of candidate DNNs, it is inexpensive to sweep over different network selection schemes to populate the optimization process with more data points. This *fine-tuning* step allows our algorithm to reach the near-optimal region *faster* compared to design space-unaware BO.

3. Our methodology identifies designs that outperform existing resource-constrained adaptive DNNs by up to 6$\times$ in terms of minimum energy per image and by up to 31.13% in terms of accuracy improvement, when tested on a commercial NVIDIA mobile board and the CIFAR-10 dataset.

The remainder of this Chapter is organized as follows: Section 4.2 provides some background material on adaptive DNNs. Section 4.3 details the proposed methodology, Section 4.4 presents the experimental setup and results, and Section 4.5 provides the discussion.

4.2 Background: Adaptive DNNs

Prior work has proposed adaptive DNNs [127], *i.e.*, systems of DNNs with different accuracy and computation characteristics, where a selection scheme *adaptively* selects the network to be evaluated for each input image [7]. Let us consider the case with M DNNs to choose from, *i.e.* $N_1, ..., N_i, ..., N_M$, as shown in Figure 4.1. For an input data item $x \in \mathcal{X}$, we denote the predictions of each i-th DNN as $N_i(x) \in R^K$, which represents the probabilities of input x belonging to each of the K classes (*i.e.*, number of K classes in the classification problem). The performance with respect to the classification task corresponds to loss $L(\tilde{y}(x), y)$, where $\tilde{y}(x)$ and $y \in \{1, 2, \ \ldots, K\}$ are the predicted and true labels, respectively; $\tilde{y}_i(x)$ is defined as the most probable class based on the current prediction, *i.e.*, $\tilde{y}_i(x) = \ \text{argmax}\ N_i(x)$. As previously discussed (Equation 2.1), L_i corresponds to the loss per network N_i. For notation brevity, here we write the loss as 0-1 loss $L(\tilde{y}_i(x), y) = \mathbb{1}_{\tilde{y}_i(x) \neq y}$, *i.e.*, whether the label $\tilde{y}_i(x)$ predicted from N_i fails to match the true label y. Once again, the hardware (cost) term per network i is $C_i = C(N_i(x))$.

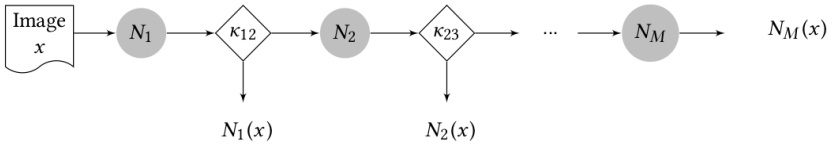

Figure 4.1: Classifying an image x using adaptive neural networks comprised of M DNNs. The system evaluates N_i first, and based on a decision function $\kappa_{i,i+1}$ decides to use $N_i(x)$ as the final prediction or to evaluate networks in later stages.

Given an image, N_1 is executed first. Next, a decision function κ is evaluated to determine whether the classification decision from N_1 should be returned as the final answer or the next network N_2 should be evaluated. In general, we denote decision function $\kappa_{ij} : N_i(x) \rightarrow \{0, 1\}$ that provides "confidence feedback" and decides between exiting at state N_i (*i.e.*, $\kappa_{ij} = 0$) or continuing at subsequent stage N_j (*i.e.*, $\kappa_{ij} = 1$).[1]

An effective choice for the κ_{ij} function is a threshold-based formulation [92, 120], where we define as score margin SM the distance between the largest and the second largest value of the vector $N_i(x) = (N_i^1(x), N_i^2(x), ..., N_i^K(x))$ at the output of network N_i. Intuitively, the more "confident" the CNN is about its prediction, the larger the value of the predicted output, thus the larger the value of SM (since $N_i(x)$'s sum to 1). For the case of $j = i + 1$, let's denote $\theta_{i,i+1}$ the threshold value for the decision function $\kappa_{i,i+1}$ between a network N_i and a network N_{i+1}. If SM in the output of N_i is larger than $\theta_{i,i+1}$, the inference result from N_i is considered correct, otherwise N_{i+1} is evaluated next, *i.e.*, $\kappa_{i,i+1}(N_i(x)|\theta_{i,i+1}) = \mathbb{1}_{[SM_i \geq \theta_{i,i+1}]}$

Key insight: The goal of the network selection problem in adaptive DNNs is to select functions κ that

[1] Without loss of generality, we show decisions between two subsequent stages, *i.e.*, $j = i + 1$, for visualization and notation simplicity. However, N_j could be any other network in the design, *i.e.*, $j \in M \wedge i \neq j$.

yields the optimal loss $\mathcal{L}$ with respect to the learning task, subject to the average per-image hardware cost term C. We make the following **key** observation: this design goal is similar to the design of DNNs with HPO methods. Thus, we can incorporate the adaptive-selection decisions directly into the hardware-constrained AutoML formulation (Equation 2.1). To the best of our knowledge there is no global, systematic methodology for the hardware-constrained HPO of adaptive DNNs, since adaptive approaches rely on off-the-shelf DNNs, without jointly optimizing the execution scheme and the DNNs.

4.3 Proposed methodology: adaptive DNNs as an AutoML problem

We directly incorporate the network-level decision into an HPO formulation.[2] That is, the loss and hardware-related terms from Equation 2.1 can be expressed as a function of the two networks (N_1, N_2) and the network-level decision function κ_{12}. We can define design space $\mathcal{Z}$ that consists of the set of hyperparameters for each network a_i and hyperparameters θ's of the decision function κ_{12}, *i.e.*, $z = (a_1, a_2, \theta_{12}), z \in \mathcal{Z}$. Hence, the energy consumed and the loss when classifying an input image $x \in \mathcal{X}$ with each neural network N_i is $E(N_i(x|a_i))$ and $L_i(\tilde{y}_i(x|a_i), y))$, respectively.

Our AutoML design goal remains the same: we want to minimize the expected loss (classification error) subject to the hardware constraint, *e.g.*, the average energy consumption per image E_T. Formally, we adapt Equation 2.1 and we write:

$$\begin{aligned} \min_{z=(a_1,a_2,\theta_{12})} \quad & \mathbb{E}_{(x,y)\sim\mathcal{X}\times\mathcal{Y}}\Big[\Big(1-\kappa_{12}(N_1(x|a_1)|\theta_{12})\Big)\cdot\Big(L_1\big(\tilde{y}_1(x|a_1),y\big)-L_2\big(\tilde{y}_2(x|a_2),y\big)\Big)\Big] \\ s.t. \quad & \mathbb{E}_{x\sim\mathcal{X}}\big[E_2\big(N_2(x|a_2)\big)\cdot\kappa_{12}\big(N_1(x|a_1)|\theta_{12}\big)+E_1\big(N_1(x|a_1)\big)\big]\leq E_T \end{aligned} \tag{4.1}$$

Note that in Equation 4.1 the overall energy and accuracy explicitly depend on the DNN designs (a_i). Moreover note that, unlike prior art that resorts in exhaustive (offline) or iterative (online) methods to find a value for the threshold [92, 120], we directly incorporate the θ values as hyperparameters to be co-optimized alongside the DNN design. Last, our formulation is generic, allowing us to consider different design paradigms, such as the case where all networks execute locally (single device) [92, 120], or when some of the networks are executed on remote servers [69], as discussed next.

Loss-constrained energy minimization: We also formulate the orthogonal problem of minimizing the average energy, while not exceeding a maximum accuracy degradation ΔL_T, *i.e.*, compared to case of only using the larger network, the error rate difference should not be greater than ΔL_T:

$$\begin{aligned} \min_{z=(a_1,a_2,\theta_{12})} \quad & \mathbb{E}_{x\sim\mathcal{X}}\Big[E_2\big(N_2(x|a_2)\big)\cdot\kappa_{12}\big(N_1(x|a_1)|\theta_{12}\big)+E_1\big(N_1(x|a_1)\big)\Big] \\ s.t. \quad & \mathbb{E}_{(x,y)\sim\mathcal{X}\times\mathcal{Y}}\Big[\Big(1-\kappa_{12}\big(N_1(x|a_1)|\theta_{12}\big)\Big)\cdot\Big(L_1\big(\tilde{y}_1(x|a_1),y\big)-L_2\big(\tilde{y}_2(x|a_2),y\big)\Big)\Big]\leq\Delta L_T \end{aligned} \tag{4.2}$$

[2] For notation simplicity, we consider $M = 2$ neural networks; we extend our analysis beyond the two-network case in [110].

Local execution: $P(N_i(x))$ is the power required when evaluating a neural network N_i an input image $x \in \mathcal{X}$. Similarly, we denote the runtime per image evaluation as $R(N_i(x))$. The expected (per image) energy is when executing both DNNs locally is:

$$\mathbb{E}_{x\sim\mathcal{X}}\Big[P_2\big(N_2(x|a_2)\big)\cdot R\big(N_2(x|a_2)\big)\cdot\kappa_{12}\big(N_1(x|a_1)|\theta_{12}\big)+P_1\big(N_1(x|a_1)\big)\cdot R\big(N_1(x|a_1)\big)\Big] \tag{4.3}$$

Remote execution of larger network: We also consider the case where some of the images $x \in \mathcal{X}$ are sent to the server over the network and the result is being communicated back, *i.e.*, N_1 executes on the edge and N_2 on the server. The expected energy is:

$$\mathbb{E}_{x\sim\mathcal{X}}\Big[P_{idle}\cdot\Big(R_{server}\big(N_2(x|a_2)\big)+\tau_{comm}\Big)\cdot\kappa_{12}\big(N_1(x|a_1)|\theta_{12}\big)+P_1\big(N_1(x|a_1)\big)\cdot R_{edge}\big(N_1(x|a_1)\big)\Big] \tag{4.4}$$

where R_{edge} and R_{server} is the runtime when executing on the edge device and the server, respectively, and P_{idle} is the idle power of the edge device while waiting for the result (for duration τ_{comm}). Please note that we assume ideal network buffers with a constant communication cost τ_{comm}, since networking effects fall outside the scope of our analysis.[3]

Enhancing Bayesian optimization: Equations 4.1 and 4.2 are variants of the main AutoML problem 2.1, we can therefore employ BO to efficiently solve them. We summarize the methodology steps in Algorithm 1. To enhance the efficiency of the BO, we make the following observation: *if we freeze the sizing of the networks considered in each outer iteration of Algorithm 1, we can cheaply fine-tune across the decision functions* κ. This step (lines 12-16), which corresponds to sweeping across θ values, has negligible complexity compared to the overall optimization overhead.

The benefit of the κ-based fine-tuning is twofold. First, in earlier stages of BO more data are being appended to the observation history $\mathcal{D}$ which improves the convergence of BO. Second, in the later states of BO, the κ-based optimization serves as fine-tuning around the near-optimal region. Effectively, our methodology combines the design space *exploration* properties inherent to BO-based methods, and the *exploitation* scope of optimizing only over the κ functions. As confirmed in our results, Algorithm 1 improves upon the designs considered during BO.

4.4 Experimental results

Experimental setup: To enable the design of adaptive DNNs via BO, we implement the key steps of Algorithm 1 on top of the `Spearmint` tool [106]. As embedded board we use NVIDIA Tegra TX1, on which we deploy the candidate networks N_i's and we measure their energy, power, and runtime values. As a

[3] Recent work by Bhardwaj *et al.* on distributed DNN inference provides a comprehensive experimental analysis when deploying DNN models on Internet-of-Things (IoT) nodes [6].

Algorithm 1 Designing adaptive DNNs via Bayesian optimization

Input: Num. iterations D_{max}, Number of networks M ($M = 2$ network case), Constraint C_T
Output: Optimizer z^* of adaptive neural networks

for $d = 1, 2, ..., D_{max}$ **do**
  $\mathcal{M} \leftarrow$ fit models on data so far $\mathcal{D}$
  $z_d \leftarrow \arg\max_{z \in \mathcal{Z}} q(z, \mathcal{M})$ // *acquisition function max.*
  // *Training and profiling each network* N_i
  $L_1(a_1^d), L_2(a_2^d) \leftarrow$ train networks N_1 and N_2
  // *power, runtime measurements on device*
  $E_1(a_1^d), E_2(a_2^d) \leftarrow$energy of N_1 and N_2
  // *Evaluate accuracy and energy consumption*
  $u^d, v^d \leftarrow$ evaluate obj. term and const. term
  $\mathcal{D} = \mathcal{D} \cap \{z^d, u^d, v^d\}$
  // κ_{12} *function fine-tuning*
  for $\theta'_{i,i+1} = 0, ..., 1.0$ **do**
    $z^d \leftarrow (a_1^d, a_2^d, \theta'_{12})$
    $u', v' \leftarrow$ evaluate obj. term and const. term
    $\mathcal{D} = \mathcal{D} \cap \{z', u', v'\}$
  end for
end for
return $z^* \leftarrow \arg\max_z \{u^1, ..., u^D\}$ *s.t.* $v^* \leq C_T$

representative comparison with prior art [92], we consider a two-network system with CaffeNet as N_1 and VGG-19 as N_2 for image classification on CIFAR-10. For all the considered cases, we use Algorithm 1 for 50 function evaluations to optimize the design of the CaffeNet network and the network selection scheme. That is, we vary the DNN design choices around the nominal CaffeNet hyperparameters [56], *i.e.*, for the convolution layers we vary the number of feature maps (32-448) and the kernel size (2-5), and for the fully connected layers the number of units (500-4000).

We consider the following **test cases** of adaptive image classification: (*i*) all DNNs execute locally on the mobile system and we denote this case as *local*; (*ii*) only N_1, *i.e.*, the less complex network is deployed on the mobile system (edge node), while the more accurate networks execute on a server (*remote* execution). For every image item, the decision function selects whether to use the local prediction or to communicate the image to the server and receive the result of the more complex network. We compute the communication time to transfer `jpeg` images and to receive the classification results back over two types of connectivity, via Ethernet and over WiFi, which we denote as *Ethernet* and *wireless* respectively.

Adaptive DNNs design: We first demonstrate the advantages that AutoML offers in this design space. For both these cases of *local* and *remote* execution with two networks, we employ grid search and we plot the obtained error-energy pairs in Figure 4.2 (left and right plot, respectively). We highlight this trade-off between accuracy and energy by drawing the Pareto front (green line).[4] We make the following observations:

[4]This visualization is insightful since the constrained optimization problem under consideration can be equivalently viewed as a multi-objective function, by writing the constraint term as the Lagrangian (*e.g.*, such formulation is used in [7]).

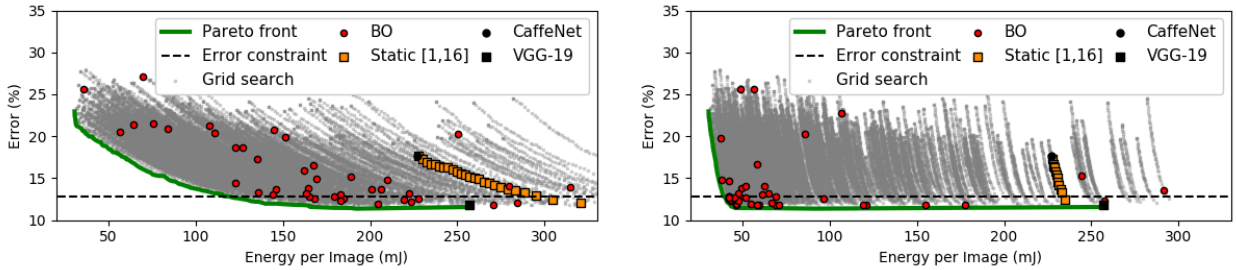

Figure 4.2: Energy minimization under maximum error constraint. Bayesian optimization considers configurations (red circles) around the near-optimal region, while significantly outperforming static-design systems (orange squares). Left: Embedded (local) execution for both networks. Right: An edge-server energy-minimization design paradigm.

first, note how far to the left the Pareto front (green line) and the the configurations considered by BO (red circles) are compared to the static optimization designs (orange squares), as well as the monolithic CaffeNet and VGG-19 networks (black markers). This captures the headroom that is available for significant reduction in energy consumption (as discussed next). *Second*, note how close to the Pareto-front red circles are, showing the effectiveness of BO at identifying the near-optimal region.

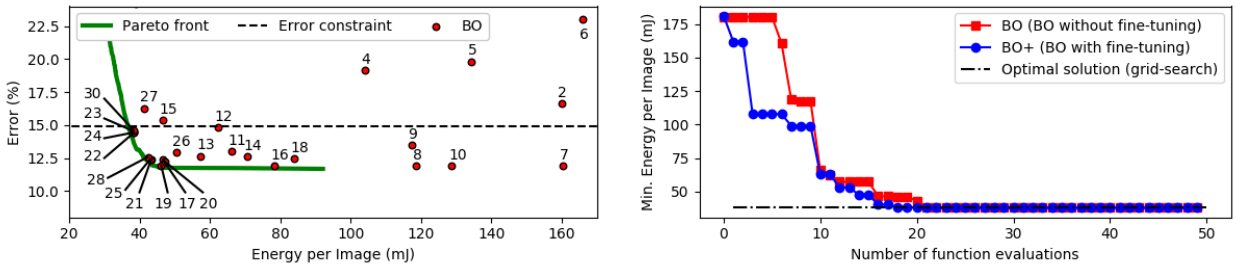

Figure 4.3: Bayesian optimization for minimum energy under error constraints in the edge-server design. The method progressively evaluates designs closer to the Pareto front. The near-optimal region is reached within 22 function evaluations. Left: Sequence of configurations selection. Right: Best solution against the number of function evaluations.

Effectiveness of BO: In Figure 4.3 we visualize the BO progress for the edge-server case. First, we enumerate the first 30 function evaluations (Figure 4.3, left), where we observe that the near-optimal region is reached with 22 steps. Next, we assess the advantage that the κ-based fine-tuning step offers. In (Figure 4.3, right), we show the minimum constraint-satisfying energy achieved during BO without (red line) and with (blue line) this step employed during the optimization. We denote these methods as BO and BO$^+$, respectively. Indeed, we observe that the fine-tuning step enhances the optimization process towards reaching the near-optimal region (grid search, dotted line) faster.

Adaptive DNNs evaluation: Next, for all three design practices, *i.e.*, *local*, *Ethernet*, and *wireless*, we solve both a constrained and an over-constrained case for both the error-constrained energy minimization

(Equation 4.2) and the energy-constrained error minimization (Equation 4.1) problems. We plot the error on the validation set and energy per-image in Figure 4.4 of the following DNN design methods: BO, BO$^+$, grid search, the best previously published work that treats the DNNs as blackboxes [92] (denoted as *static*), and using either N_1 (CaffeNet) or N_2 (VGG-19) on their own. We report the constraints per case in the parentheses on the x axis (the error percentage value corresponds to the maximum accuracy degradation allowed compared to always using VGG-19, *i.e.*, the B value in Equation 4.2). We also report the test error of each method in Table 4.1.

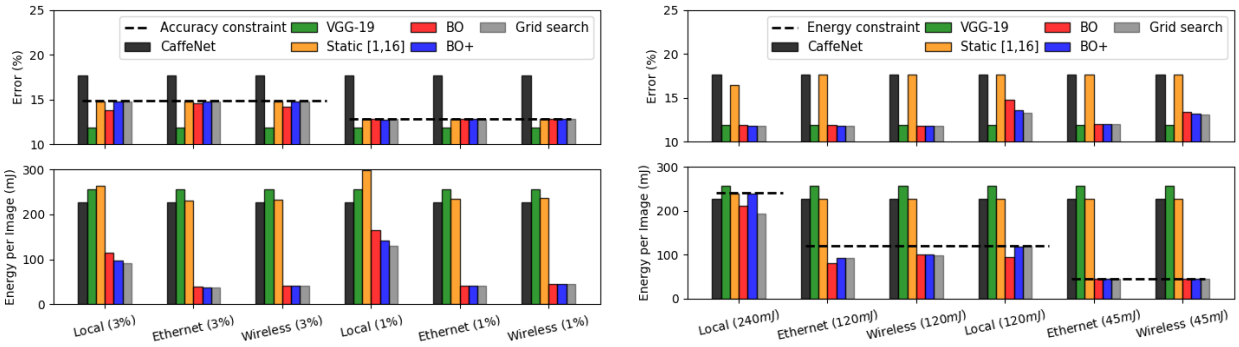

Figure 4.4: Assessing the effectiveness of the proposed methodology across different design paradigms and both constrained and over-constrained cases (the constraint values per case are given in the parentheses on the x-axis labels). We observe that our methodology BO$^+$ (blue) successfully approaches the grid-search solution (gray), while always outperforming the best solution achieved by existing static-design methods (orange). Left: Energy minimization under maximum error constraints. Right: Error minimization under maximum energy constraints.

Table 4.1: Hyperparameter optimization results on the test set.

	Minimum energy (mJ) achieved under allowed accuracy degradation constraint (given in parenthesis).					
Method	Local (3%)	Ethernet (3%)	Wireless (3%)	Local (1%)	Ethernet (1%)	Wireless (1%)
Static [7, 92]	256.09	230.05	231.15	292.25	233.50	235.97
BO	114.43	38.28	41.49	163.15	41.85	45.54
BO$^+$	99.35	38.00	40.50	148.26	41.68	45.54
Grid	93.43	38.00	40.50	123.57	41.68	45.49
	Minimum error (%) achieved under energy constraint (given in parenthesis).					
Method	Local (240mJ)	Ethernet (120mJ)	Wireless (120mJ)	Local (120mJ)	Ethernet (45mJ)	Wireless (45mJ)
Static [7, 92]	17.30	18.44	18.44	18.44	18.44	18.44
BO	12.67	12.77	12.74	20.69	12.83	14.42
BO$^+$	12.66	12.70	12.74	16.30	12.83	13.98
Grid	12.58	12.58	12.58	14.14	12.82	13.98

Based on the results, we can make several observations: (*i*) **Suboptimality of prior work**: As discussed previously, the energy consumption of CaffeNet and VGG-19 allows for only a small energy-saving headroom to exploit via the *static* method. In fact, for *local* energy minimization with 1% maximum accuracy degradation allowed (similar constraint as in [92]), statically designed adaptive DNNs will be forced to always use VGG-19 for the final prediction, while wastefully evaluating CaffeNet. This results in larger energy consumption than using VGG-19 by itself. (*ii*) **Effectiveness proposed method**: We observe

that in all the considered cases the proposed BO$^+$ method closely matches the result identified by grid search. In general, our methodology identifies designs that outperform static methods [92] by up to 6× in terms of minimum energy under accuracy constraints and by up to 31.13% in terms of error minimization under energy constraints.

Moreover, (*iii*) **κ-based fine-tuning**: As expected, the fine-tuning step that we enhances the BO. In particular, BO$^+$ leads to further energy minimization under accuracy constraints by 13.18% and to error minimization under energy constraints by 21.22%, compared to the result of BO without fine-tuning. (*iv*) **Local versus remote execution**: we observe that executing some of DNNs comprising the adaptive neural network remotely allows for more energy-efficient image classification, compared to executing everything locally at the same level of accuracy. That is, using an edge-server design, where only the smallest CNN executes locally, allows for energy reduction of 2.96× compared to executing all networks locally and for the same error constraint.

Table 4.2: Designing three-network adaptive DNNs: Hyperparameter optimization results on the test set.

Method	Minimum energy (mJ) under accuracy degradation constraint: Local (3%)	Minimum error (%) under energy constraint: Local (120mJ)
Static [7, 92]	268.43	41.25
BO	137.36	17.94
BO$^+$	113.59	15.06
Grid	108.45	14.98

Exploring three-network adaptive DNNs: Finally, we evaluate the proposed method on a three-network case. We summarize the results in Table 4.2. Once again, we observe that, compared to static methods, our methodology closely matches the results obtained by grid search. More specifically, in the case of energy minimization, the solution reached by BO^+ is only 4.74% away from the optimal grid search-based design, while outperforming best previously published static methods [92] by 2.47× in terms of energy minimization.

4.5 Discussion

In this Chapter, we introduce an efficient HPO methodology to design hardware-constrained adaptive DNNs based on BO. The key novelty in our work is that both the DNN architectures and the selection scheme are treated as hyperparameters that are globally (jointly) optimized. This allows us to identify designs that outperform existing adaptive DNNs by up to 6× in terms of minimum energy per image under accuracy constraints and by up to 31.13% in terms of error minimization under energy constraints.

Finally, we study two image classification practices, *i.e.*, classifying all images locally versus over the cloud under energy and communication constraints.

To motivate an interesting direction for future work, we note that the methodologies presented in both this (Chapter 4) and the previous chapter (Chapter 3) are based on sequential network samples. That is, in each iteration the hyperparameter optimizer suggests a (set of) candidate DNN designs which are trained and evaluated on a validation set from scratch. In the context of NAS methods, recent AutoML work has shown that *one-shot* methodologies can significantly reduce the search cost [79]. In the following Chapter 5, we delve into *one-shot* AutoML: we identify the key bottleneck in existing methodologies and we achieve new state-of-the-art results for Mobile AutoML applications.

Chapter 5

Efficient Single-Path Neural Architecture Search

Neural Architecture Search (NAS) has drawn an unprecedented surge of interest from the AutoML community in the recent months. *One-shot* NAS methodologies have been recently shown to significantly reduce the NAS search cost compared to their *standalone* counterparts [79]. In this Chapter, we delve into *one-shot* NAS formulations and we identify their key limitations. In a novel departure from existing (*multi-path*) assumptions on how the different DNN architectures are encoded across separate paths in the *one-shot* model, we introduce a state-of-the-art *single-path* NAS encoding of the search space.

5.1 Chapter overview

Earlier work on AutoML, including our Bayesian optimization methodologies introduced in Chapters 3-4, have been originally studied in the context of sequential point evaluations. With the recent surge of interest in Neural Architecture Search (NAS), the earlier NAS methods based on either evolutionary algorithms (EA) or reinforcement learning (RL) adhere to this sequential setting. That is, the AutoML search progresses over evaluations of *standalone* DNNs and the networks that have already been evaluated are used either for mutations over the EA population [97] or to train the RL agent using a policy gradient algorithm [138].

However, these methods are not ideally suited for optimizing problems where point evaluations are expensive to obtain [61]. Indeed, architecture search is in essence an optimization problem, *i.e.*, aiming to find the DNN with the lowest validation error: there is no explicit need to maintain a notion of mutations for EA or to solve the credit assignment for RL [61]. Since either methods could be fundamentally more difficult problems than optimization [57], these approaches require thousands of candidate DNNs to be trained [128]. Consequently, updating the RL controller or the EA population poses prohibitive computational challenges and thousands of candidate DNNs need to be trained.

The aforementioned challenge has propelled the AutoML community to investigate novel formulations

that reduce the computational burden of *standalone* NAS methods. In a seminal work, Liu *et al*. have shown that formulating the NAS problem in a differentiable manner excels in discovering high-performance DNN architectures [79], while being orders of magnitude faster than previous *standalone* techniques. The main idea of *one-shot* NAS is to relax the design of DNNs (Equation 2.1) to a differentiable operation/path selection problem: first, an over-parameterized, *multi-path* supernet is constructed, where, for each layer, every candidate operation is added as a *separate* trainable path, as illustrated in Figure 5.1 (left). Next, *multi-path* NAS formulations solve for the paths that yield the optimal architecture. This insight has given rise to a plethora of novel *one-shot* methodologies [128, 12, 132, 84].

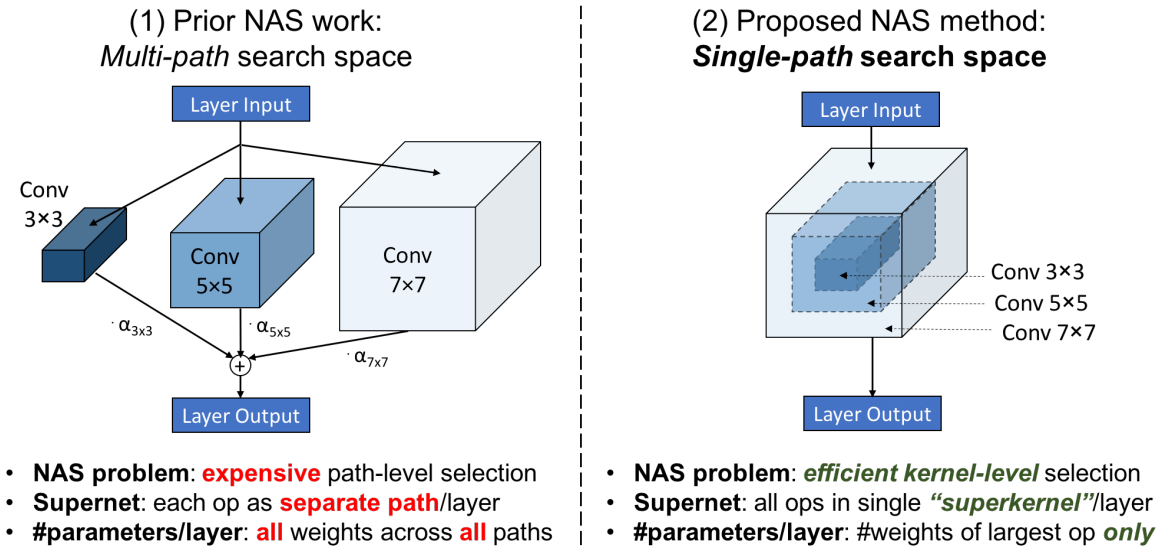

Figure 5.1: Our method directly optimizes for the subset of convolution kernel weights and searches over an over-parameterized "**superkernel**" in each DNN layer (right). This **novel view** of the design space eliminates the need for maintaining separate paths for each candidate operation, as in previous *multi-path* approaches (left).

As expected, naïvely branching out all paths is inefficient due to an intrinsic limitation: the number of trainable parameters that need to be maintained and updated during the search grow linearly with respect to the number of candidate operations per layer [1]. To tame the memory explosion introduced by the *multi-path* supernet, current methods employ creative "workaround" solutions: *e.g.*, searching on a proxy dataset (subset of ImageNet [128]), or employing a memory-wise scheme with only a subset of paths being updated during the search [12]. Nevertheless, these techniques remain considerably costly, with an overall computational demand of at least 200 GPU-hours.

This observation motivates a novel exploration in this thesis based on the following question: "can we match the performance of existing *one-shot* NAS in terms of accuracy of the identified DNN design, while

reducing the search cost from days down to **only a few hours**?" A critical challenge in this endeavor is that we need to rethink how the NAS search space can be represented with the *one-shot* over-parameterized supernet in a more efficient way.

5.1.1 Key novelty: from *multi-* to *single-path* NAS formulations

To address the suboptimality of prior work, we propose *Single-Path* NAS. Our **key insight** is illustrated in Figure 5.1 (right). We build upon the observation that different candidate convolutional operations in NAS can be viewed as subsets of a **single "superkernel"**. Without having to choose among different paths/operations as in *multi-path* methods, we instead solve the NAS problem as *finding which subset of kernel weights to use in each DNN layer*. By sharing the kernel weights, we encode all candidate NAS operations into a single **"superkernel"**, *i.e.*, with a single path, for each layer of the *one-shot* NAS supernet.

5.1.2 Contributions and Chapter organization

To the best of our knowledge, our *Single-Path* NAS methodology brings the following novel contributions:

1. ***Single-Path* NAS**: We propose a novel view of the one-shot, supernet-based design space, hence drastically decreasing the number of trainable parameters. *Single-Path* NAS is the *first* work to formulate the NAS problem as finding the subset of kernel weights in each DNN layer [113, 114].

2. **State-of-the-art results**: We achieve an overall search cost of only **8 epochs**, *i.e.*, **3.75 hours** on TPUs (30 TPU-hours), which is up to **5,000× faster** compared to prior work. *Single-Path* NAS achieves state-of-the-art top-1 accuracy on ImageNet with mobile latency on-par with previously best Mobile AutoML methodologies ($\approx 80ms$ on a Pixel 1).

This Chapter is organized as follows: Section 5.2 presents our novel view of the design space and introduces the *Single-Path* NAS formulation. Section 5.3 and Section 5.4 provide the experimental results and discussion, respectively.

5.2 Proposed *Single-Path* NAS

To simplify notation and to illustrate the **key idea**, without loss of generality, we show the case of choosing between a 3×3 or a 5×5 kernel for an MBConv layer. Let us denote the weights of the two candidate kernels as $\mathbf{w}_{3\times3}$ and $\mathbf{w}_{5\times5}$, respectively. As shown in Figure 5.2, we observe that the weights of the 3×3 kernel can be viewed as the *inner* core of the weights of the 5×5 kernel, while "zeroing" out the weights

of the "*outer*" shell. We denote this (*outer*) subset of weights (that does not contribute to output of the 3×3 kernel but only to the 5×5 kernel), as $\mathbf{w}_{5\times5\backslash3\times3}$. Hence, the NAS architectural choice of using the 5×5 convolution corresponds to using both the *inner* $\mathbf{w}_{3\times3}$ weights and the *outer* shell, *i.e.*, $\mathbf{w}_{5\times5} = \mathbf{w}_{3\times3} + \mathbf{w}_{5\times5\backslash3\times3}$ (Figure 5.2).

We can therefore encode the NAS decision directly into the **superkernel** of an MBConv layer as a function of kernel weights as follows:

$$\mathbf{w}_k = \mathbf{w}_{3\times3} + \mathbb{1}(\text{use } 5\times5) \cdot \mathbf{w}_{5\times5\backslash3\times3} \tag{5.1}$$

where $\mathbb{1}(\cdot)$ is the indicator function that encodes the architectural NAS choice, *i.e.*, if $\mathbb{1}(\cdot) = 1$ then $\mathbf{w}_k = \mathbf{w}_{3\times3} + \mathbf{w}_{5\times5\backslash3\times3} = \mathbf{w}_{5\times5}$, else $\mathbb{1}(\cdot) = 0$ then $\mathbf{w}_k = \mathbf{w}_{3\times3}$.

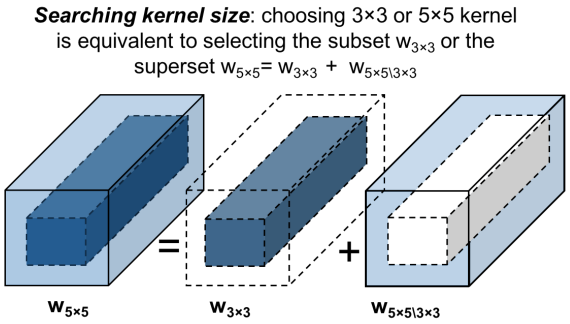

Figure 5.2: Encoding NAS kernel-level decisions into the **searchable superkernel**.

Trainable indicator/condition function: While the indicator function encodes the NAS decision, a critical choice is how to formulate the condition over which the $\mathbb{1}(\cdot)$ is evaluated. Our intuition is that, for an indicator function that represents whether to use the subset of weights, its condition should be *directly a function of the subset's weights*. Thus, our goal is to define an "importance" signal of the subset weights that intrinsically captures their contribution to the overall DNN loss. We draw inspiration from weight-based conditions that have been successfully used for quantization-related decisions [27] and we use the *group Lasso term*. Specifically, for the indicator related to the $\mathbf{w}_{5\times5\backslash3\times3}$ "outer shell" decision, we write the following condition:

$$\mathbf{w}_k = \mathbf{w}_{3\times3} + \mathbb{1}(\left\|\mathbf{w}_{5\times5\backslash3\times3}\right\|^2 > t_{k=5}) \cdot \mathbf{w}_{5\times5\backslash3\times3} \tag{5.2}$$

where $t_{k=5}$ is a latent variable that controls the decision (*e.g.*, a threshold value) of selecting kernel 5×5. The threshold will be compared to the Lasso term to determine if the *outer* $\mathbf{w}_{5\times5\backslash3\times3}$ weights are used to the overall convolution. It is important to notice that, instead of picking the thresholds (*e.g.*, $t_{k=5}$) by hand, we seamlessly treat them as trainable parameters to learn via gradient descent. To compute the gradients for thresholds, we relax the indicator function $g(x,t) = \mathbb{1}(x > t)$ to a sigmoid function, $\sigma(\cdot)$, when computing gradients, *i.e.*, $\hat{g}(x,t) = \sigma(x > t)$.

Searching for expansion ratio and skip-op: Since the result of the kernel-based NAS decision $\mathbf{w}_k$ (Equation 5.2) is a convolution kernel itself, we can in turn apply our formulation to also encode NAS decisions for the expansion ratio of the $\mathbf{w}_k$ kernel. As illustrated in Figure 5.3, the channels of the depthwise

convolution in an MBConv-$k \times k$-3 layer with expansion ratio $e = 3$ can be viewed as using one half of the channels of an MBConv-$k \times k$-6 layer with expansion ratio $e = 6$, while "zeroing" out the second half of channels $\{\mathbf{w}_{k,6\backslash 3}\}$. Finally, by "zeroing" out the first half of the output filters as well, the entire **superkernel** contributes nothing if added to the residual connection of the MBConv layer: *i.e.*, by deciding if $e = 3$, we can encode the NAS decision of using, or not, only the "skip-op" path. For both decisions over $\mathbf{w}_k$ kernel, we write:

$$\mathbf{w} = \mathbb{1}(\|\mathbf{w}_{k,3}\|^2 > t_{e=3}) \cdot (\mathbf{w}_{k,3} + \mathbb{1}(\left\|\mathbf{w}_{k,6\backslash 3}\right\|^2 > t_{e=6}) \cdot \mathbf{w}_{k,6\backslash 3}) \tag{5.3}$$

Hence, for input $\mathbf{x}$, the output of the i-th MBConv layer of the network is:

$$o^i(\mathbf{x}) = \text{conv}(\mathbf{x}, \mathbf{w}^i | t^i_{k=5}, t^i_{e=6}, t^i_{e=3}) \tag{5.4}$$

Searchable MBConv kernels: Each MBConv uses 1×1 convolutions for the point-wise (first) and linear stages, while the kernel-size decisions affect only the (middle) $k \times k$ depthwise convolution (Figure 5.4). To this end, we use our **searchable** $k \times k$ depthwise kernel at this middle stage. In terms of number of channels, the depthwise kernel depends on the point-wise 1×1 output, which allows us to directly encode the expansion ratio e at the middle stage as well: by setting the point-wise 1×1 output to the maximum candidate expansion ratio, we can instead solve for which of them not to "zero" out at the depthwise (middle) state. In other words, we directly use our **searchable** depthwise convolution **superkernel** to effectively encode the NAS decision for the expansion ratio. Hence, our *single-path*, convolution-based formulation can sufficiently capture any MBConv type (*e.g.*, MBConv-3 $\times$ 3-6, MBConv-5 $\times$ 5-3, *etc.*) in the MobileNetV2-based design space (Figure 5.4).

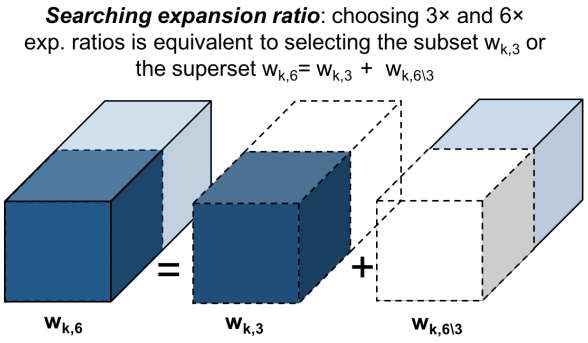

Figure 5.3: Encoding *expansion ratio* decisions into the **searchable superkernel**.

5.2.1 *Single-path vs.* existing *multi-path* assumptions

As a reminder, in Chapter 2 we briefly illustrated how *multi-path* existing methods [12, 79, 128] solve the differentiable NAS problem, where the output of each layer i is a (weighted) sum defined over the output of N different paths (Equation 2.2). Hence, it is easy to see how our novel *single-path* view is advantageous, since the output of the convolution at layer i of our search space is *directly a function of the weights of our*

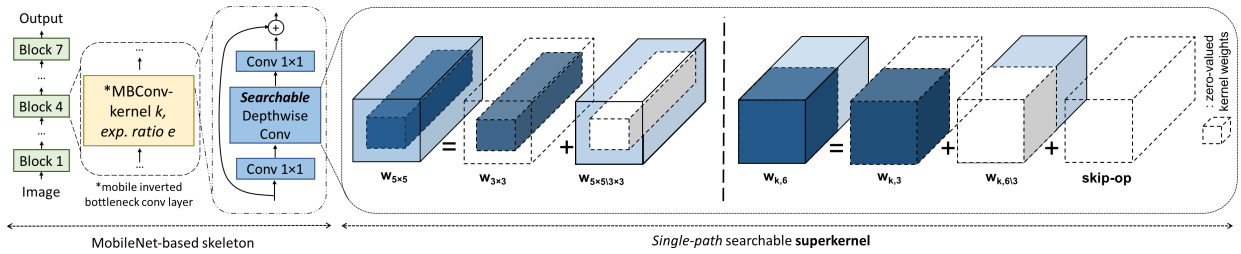

Figure 5.4: *Single-path* NAS builds upon the MobileNetV2-based search space [122] to identify the mobile inverted bottleneck convolution (MBConv) per layer (left). Our *one-shot supernet* encapsulates all possible NAS architectures in the search space, *i.e.*, different kernel size (middle) and expansion ratio (right) values, without the need for appending each candidate operation as a separate path. *Single-Path* NAS directly searches over the weights of the per-layer **searchable "superkernel"** that encodes all MBConv types.

single over-parameterized kernel (Equation 5.4):

$$o^i_{single-path}(\mathbf{x}) = o^i(\mathbf{x}) = \text{conv}(\mathbf{x}, \mathbf{w}^i | t^i_{k=5}, t^i_{e=6}, t^i_{e=3}) \tag{5.5}$$

More importantly, with our *single-path* formulation, the overall network loss is directly a function of the **"superkernel"** weights, where the learnable kernel- and expansion ratio-related threshold variables, $\mathbf{t}_k$ and $\mathbf{t}_e$, are directly derived as a function (norm) of the kernel weights $\mathbf{w}$. Consequently, *Single-Path* NAS formulates the NAS problem as solving *directly over the weight kernels* $\boldsymbol{w}$ *of a single-path, compact neural network*. Formally, the NAS problem becomes:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w} | \mathbf{t}_k, \mathbf{t}_e) \tag{5.6}$$

Efficiency of *Single-Path* NAS: Unlike the bi-level optimization problem in prior differentiable NAS methods (Section 2, Equation 2.3), solving our NAS formulation in Equation 5.6 is as expensive as training the weights of a single-path, **branchless**, compact neural network with vanilla gradient descent. Therefore, our formulation eliminates the need for separate gradient steps between the DNN weights and the NAS parameters. Moreover, the reduction of the trainable parameters $\mathbf{w}$ per se, further leads to a drastic reduction of the search cost down to **just a few epochs**, as our experimental results show later in Section 5.3. Our NAS problem formulation allows us to efficiently solve Equation 5.6 with batch sizes of 1024, a four-fold increase compared to prior art's search efficiency.

5.2.2 Hardware-aware NAS with differentiable runtime loss

To design hardware-efficient DNNs, the differentiable objective in Equation 5.6 should reflect both the accuracy of the searched architecture and its inference latency on the target hardware. Hence, we use a

latency-aware formulation [12, 128]:

$$\mathcal{L}(\mathbf{w}|\mathbf{t}_k, \mathbf{t}_e) = CE(\mathbf{w}|\mathbf{t}_k, \mathbf{t}_e) + \lambda \cdot \log(R(\mathbf{w}|\mathbf{t}_k, \mathbf{t}_e)) \tag{5.7}$$

The first term CE corresponds to the cross-entropy loss of the single-path model. The hardware-related term R is the runtime in milliseconds (ms) of the searched NAS model on the target mobile platform. Finally, the coefficient λ modulates the trade-off between cross-entropy and runtime.

Runtime model over the *single-path* design space: To preserve the differentiability of the objective, another critical choice is the formulation of the latency term R. Prior art has showed that the total network latency of a mobile DNN can be modeled as the sum of each i-th layer's runtime R^i, since the runtime of each operator is independent of other operators [8, 12, 128]:

$$R(\mathbf{w}|\mathbf{t}_k, \mathbf{t}_e) = \sum_i R^i(\mathbf{w}^i|\mathbf{t}_k^i, \mathbf{t}_e^i) \tag{5.8}$$

For our approach, we adapt the per-layer runtime model as a function of the NAS-related decisions $\mathbf{t}$. We profile the target mobile platform (Pixel 1) and we record the runtime for each candidate kernel operation per layer i, *i.e.*, $R^i_{3\times3,3}$, $R^i_{3\times3,6}$, $R^i_{5\times5,3}$, and $R^i_{5\times5,6}$. We denote the runtime of layer i by following the notation in Equation 5.3. Specifically, the runtime of layer i is defined first as a function of the expansion ratio decision:

$$R^i_e = \mathbb{1}(\left\|\mathbf{w}_{k,3}\right\|^2 > \mathbf{t}_{e=3}) \cdot (R^i_{5\times5,3} + \mathbb{1}(\left\|\mathbf{w}_{k,6\setminus3}\right\|^2 > \mathbf{t}_{e=6}) \cdot (R^i_{5\times5,6} - R^i_{5\times5,3})) \tag{5.9}$$

Next, by incorporating the kernel size decision, the total runtime is:

$$R^i = \frac{R^i_{3\times3,6}}{R^i_{5\times5,6}} \cdot R^i_e + R^i_e \cdot (1 - \frac{R^i_{3\times3,6}}{R^i_{5\times5,6}}) \cdot \mathbb{1}(\left\|\mathbf{w}_{5\times5\setminus3\times3}\right\|^2 > \mathbf{t}_{k=5}) \tag{5.10}$$

As in Equation 5.2, we relax the indicator function to a sigmoid function $\sigma(\cdot)$ when computing gradients. By using this model, the runtime term in the loss function remains differentiable with respect to layer-wise NAS choices. As we show in our results, the model is accurate, with an average prediction error of 1.76%.

5.3 Experimental results

5.3.1 Experimental setup

We use *Single-Path* NAS to design DNNs for image classification on ImageNet [24][1]. We use Pixel 1 as the target mobile platform. This experimental setup allows for a representative comparison with prior hardware-efficient NAS that optimize for Pixel 1 devices around a target latency of 80ms [12, 122].

[1] https://github.com/dstamoulis/single-path-nas

We implement our NAS framework in TensorFlow (TF-1.12) running on TPUs-v2 [59], following the TPU-based MnasNet documentation[2]. We use Keras to implement our trainable superkernels, where we define a custom Keras-based convolution kernel where the output is a function of both the weights and the threshold-based decisions (Equations 5.2-5.3). Our custom layer also returns the layer runtime (Equations 5.9-5.10).

5.3.2 Runtime profiling and modeling

To train the runtime model, we deploy the DNNs to the mobile device with TensorFlow TFLite. On the device, we profile runtime using the Facebook AI Performance Evaluation Platform (FAI-PEP)[3] that supports profiling for tflite models with detailed per-layer runtime breakdown. We record the runtime per layer (MBConv breakdown) by profiling DNNs with different MBConv types, *i.e.*, we obtain the $R^i_{3\times3,3}$, $R^i_{3\times3,6}$, $R^i_{5\times5,3}$, and $R^i_{5\times5,6}$ runtime values per MBConv layer i (Equations 5.9-5.10). To evaluate the runtime-prediction accuracy of the model, we generate 100 randomly designed DNNs and we measure their runtime on the device. As illustrated in Figure 5.5, our model can accurately predict the DNN runtimes: the Root Mean Squared Error (RMSE) is 1.32ms (1.76% mean prediction error).

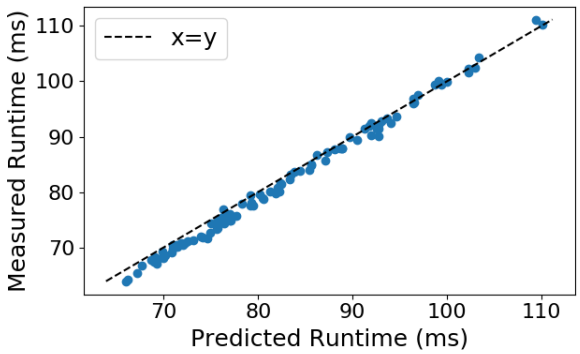

Figure 5.5: The runtime model (Equation 5.8) is accurate, with 1.76% mean prediction error.

5.3.3 State-of-the-art runtime-constrained ImageNet classification

We apply our method to design DNNs for the Pixel 1 phone with an overall target latency of 80ms. We train the derived *Single-Path* NAS model for 350 epochs, following the MnasNet training schedule [122]. We compare our method with mobile DNNs designed by human experts and state-of-the-art NAS methods in Table 5.1, in terms of classification accuracy and search cost. In terms of hardware efficiency, prior work has shown that low FLOP count does not necessarily translate to high hardware efficiency [31], we therefore evaluate the various NAS methods with respect to the inference runtime on Pixel 1 ($\leq$ 80ms).

Enabling a representative comparison: While we provide the original values from the respective papers, our goal is to ensure a fair comparison. To this end, we retrain the baseline models following the same schedule (in fact, we find that the MnasNet-based training schedule improves the top1 accuracy

[2] https://github.com/tensorflow/tpu/tree/master/models/official/mnasnet
[3] https://github.com/facebook/FAI-PEP

Table 5.1: *Single-Path* NAS achieves state-of-the-art accuracy (%) on ImageNet for similar mobile latency setting compared to previous NAS methods ($\leq$ 80*ms* on Pixel 1), with up to 5,000$\times$ reduced search cost in terms of number of epochs. *The search cost in epochs is estimated based on the claim [12] that ProxylessNAS is 200$\times$ faster than MnasNet. ‡ChamNet does not detail the model derived under runtime constraints [23] so we cannot retrain or measure the latency.

Method	Top-1 Acc (%)	Top-5 Acc (%)	Mobile Runtime (ms)	Search Cost (epochs)
MobileNetV1 [50]	70.60	89.50	113	-
MobileNetV2 1.0x [102]	72.00	91.00	75.00	
MobileNetV2 1.0x (our impl.)	73.59	91.41	73.57	
Random search	73.78 $\pm$ 0.85	91.42 $\pm$ 0.56	77.31 $\pm$ 0.9 ms	-
MnasNet-B1 [122]	74.00	91.80	76.00	40,000
MnasNet-B1 (our impl.)	74.61	91.95	74.65	
ChamNet-B [23]	73.80	–	–	240‡
ProxylessNAS-R [12]	74.60	92.20	78.00	200*
ProxylessNAS-R (our impl.)	74.65	92.18	77.48	
FBNet-B [128]	74.1	-	-	90
FBNet-B (our impl.)	73.70	91.51	78.33	
Single-Path NAS (**proposed**)	**74.96**	**92.21**	79.48	**8 (3.75 hours)**

compared to what is reported in several previous methods). Similarly, we profile the models on the same Pixel 1 device. For prior work that does not optimize for Pixel 1, we retrain and profile their model closest to the MnasNet baseline (*e.g.*, the FBNet-B and ChamNet-B networks [23, 128], since the authors use these DNNs to compare against the MnasNet model). Finally, to enable a representative comparison of the search cost per method, we directly report the number of epochs reported per method, hence canceling out the effect of different hardware systems (GPU *vs.* TPU hours).

ImageNet classification: Table 5.1 shows that our *Single-Path* NAS achieves top-1 accuracy of **74.96%**, which is the new state-of-the-art ImageNet accuracy among hardware-efficient NAS methods. Specifically, our method achieves better top-1 accuracy than ProxylessNAS by $+$**0.31**%, while maintaining on par target latency of $\leq$ 80*ms* on the same target mobile phone. *Single-Path* NAS outperforms methods in this mobile latency range, *i.e.*, better than MnasNet (+0.35%), FBNet-B (+0.86%), and MobileNetV2 (+1.37%).

NAS search cost: *Single-Path* NAS has **orders of magnitude reduced search cost** compared to all previous hardware-efficient NAS methods. Specifically, MnasNet reports that the controller uses 8k sampled models, each trained for 5 epochs, for a total of 40k train epochs. In turn, ChamNet trains an accuracy predictor on 240 samples, which assuming an aggressively fast training schedule of five epochs per sample (same as in MnasNet), corresponds to a total search cost of 1.2k epochs. ProxylessNAS reports 200$\times$ search cost improvement over MnasNet, hence the overall cost is the TPU-equivalent of 200 epochs. Finally, FBNet reports 90 epochs of training on a proxy dataset (10% of ImageNet). While the number of images per epoch is reduced, we found that a TPU can accommodate a FBNet-like supermodel with

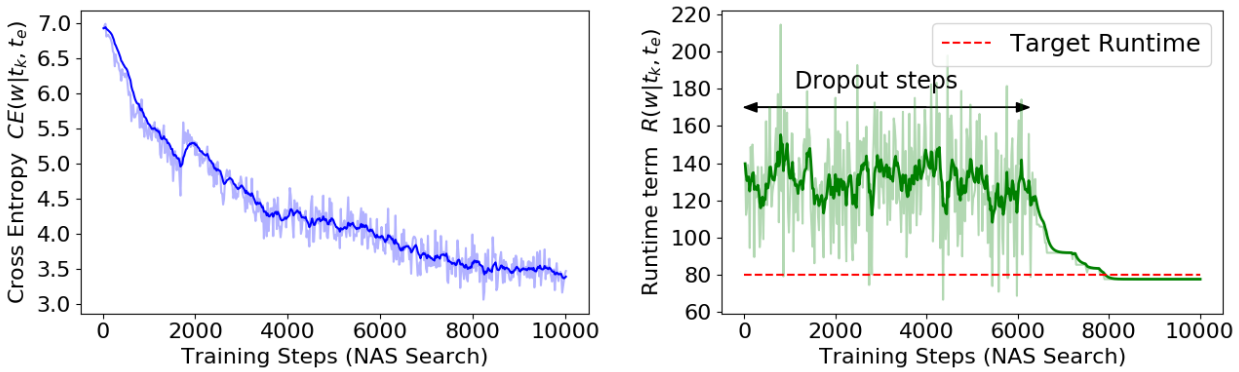

Figure 5.6: *Single-Path* NAS search progress: Progress of both objective terms, *i.e.*, cross entropy CE (left) and runtime R (right) during NAS search.

maximum batch size of 128, hence the number of steps per FBNet epoch are still 8× more compared to the steps per epoch in our method.

In comparison, *Single-Path* NAS has a total cost of eight epochs, which is **5,000**× faster than MnasNet, **25**× faster than ProxylessNAS, and **11**× faster than FBNet. We use an aggressive training schedule similar to the few-epochs schedule used in MnasNet to train the individual DNN samples [122]. We visualize the search efficiency of our method in Figure 5.6, where we show the progress of both CE and R terms. Earlier during our search (first six epochs), we employ *dropout* across the different subsets of the kernel weights (Figure 5.6, right). Dropout is a common technique in NAS methods to prevent the supernet from learning as an ensemble. That is, we randomly drop the subsets of the superkernel in our *single-path* search space. We search for $\sim 10k$ **steps** (8 epochs with a batch size of 1024), which corresponds to total wall-clock time of **3.75 hours** on a TPUv2. In particular, given than a TPUv2 has 2 chips with 4 cores each, this corresponds to a total of 30 TPU-hours.

Different channel size scaling: Next, we follow a typical analysis [12, 128], by rescaling the networks using a width multiplier [102]. As shown in Figure 5.7, we observe that our model consistently outperforms prior methods under varying runtime settings. For instance, *Single-Path* NAS with 79.48*ms* is 1.56× faster than the MobileNetV2 scaled model of similar accuracy.

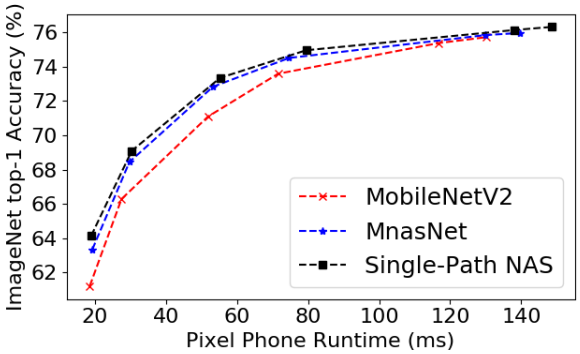

Figure 5.7: Our method outperforms MobileNetV2 & MnasNet across various size scales.

Visualization of *Single-Path* NAS DNN: Our derived DNN architecture is shown in Figure 5.8.

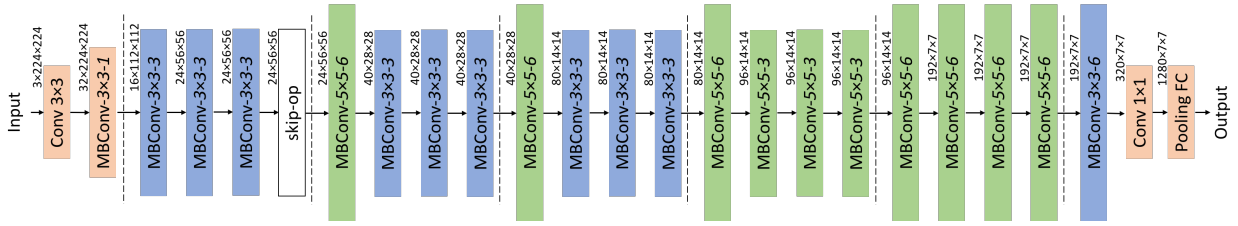

Figure 5.8: Hardware-efficient DNN found by *Single-Path* NAS, with top-1 accuracy of **74.96%** on ImageNet and inference time of 79.48*ms* on Pixel 1 phone.

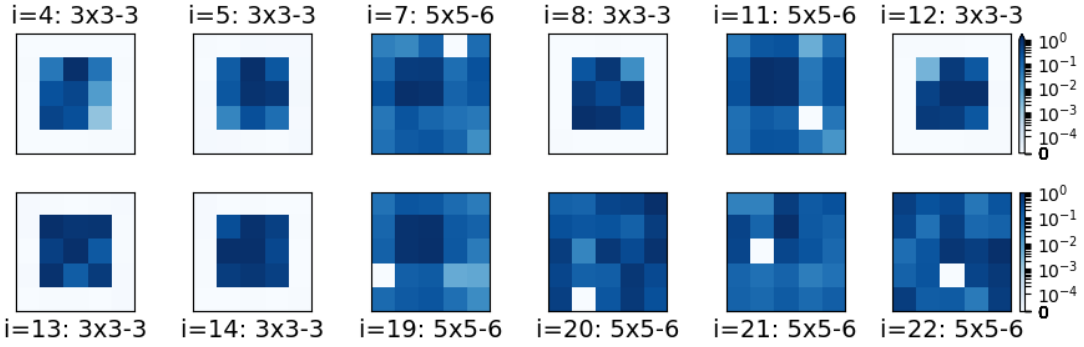

Figure 5.9: Visualization of kernel-based architectural contributions. The *standard deviation* of **superkernel** values across the kernel channels is shown in log-scale, with lighter colors indicating smaller values.

Moreover, to illustrate how the searchable superkernels effectively capture NAS decisions across subsets of kernel weights, we plot the standard deviation of weight values in Figure 5.9 (shown in log-scale, with lighter colors indicating smaller values). Specifically, we compute the standard deviation of weights across the channel-dimension for all superkernels. For various layers shown in Figure 5.9 (per i-th DNN's layer from Figure 5.8), we observe that the *outer* $\mathbf{w}_{5\times5\setminus3\times3}$ "shells" reflect the NAS architectural choices: for layers where the entire $\mathbf{w}_{5\times5}$ is selected, the $\mathbf{w}_{5\times5\setminus3\times3}$ values drastically vary across the channels. On the contrary, for all layers where 3×3 convolution is selected, the *outer* shell values do not vary significantly.

5.3.4 Ablation study: kernel-based accuracy-efficiency trade-off

Our method searches over subsets of convolutional kernel weights. Hence, we conduct experiments to highlight how kernel-weight subsets can capture accuracy-efficiency trade-off effectively. We use the MobileNetV2 macro-architecture as a backbone (we maintain the location of stride-2 layers as default). As two baseline networks, we consider the default MobileNetV2 with MBConv-3 $\times$ 3-6 blocks (*i.e.*, $\mathbf{w}_{3\times3}$ kernels for all depthwise convolutions), and a network with MBConv-5 $\times$ 5-6 blocks (*i.e.*, $\mathbf{w}_{5\times5}$ kernels).

Next, to capture the subset-based training of weights during a *Single-Path* NAS search, we consider a *DNN* with MBConv-5 $\times$ 5-6 blocks, where we compute the loss of the model over two subsets, (i) the inner

Table 5.2: Searching across subsets of kernel weights: DNNs with weight values trained over subsets of the kernels (3×3 as subset of 5×5) achieve performance (top-1 accuracy) similar to DNNs with individually trained kernels.

Method	Top-1 Acc (%)	Top-5 Acc (%)
Baseline DNN - $\mathbf{w}_{3\times3}$ kernels	73.59	91.41
Baseline DNN - $\mathbf{w}_{5\times5}$ kernels	74.10	91.67
Single-Path DNN - inference w/ $\mathbf{w}_{3\times3}$ kernels	73.43	91.42
Single-Path DNN - inference w/ $\mathbf{w}_{3\times3} + \mathbf{w}_{5\times5\setminus3\times3}$ kernels	73.86	91.72

$\mathbf{w}_{3\times3}$ weights, and (ii) by also using the remaining $\mathbf{w}_{5\times5\setminus3\times3}$ weights. For each loss computed over these subsets, we accumulate back-propagated gradients and update the respective weights, *i.e.*, gradients are being applied separately to the inner and to the entire kernel per layer. We follow training steps similar to the "switchable" training across channels as in [135] (for the remaining training hyper-parameters we use the same setup as the default MnasNet). As shown in Table 5.2, we observe the final accuracy across the kernel granularity, *i.e.*, with the inner $\mathbf{w}_{3\times3}$ and the entire $\mathbf{w}_{5\times5} = \mathbf{w}_{3\times3} + \mathbf{w}_{5\times5\setminus3\times3}$ kernels, follows an accuracy change relative to DNNs with individually trained kernels.

Such finding is significant in the context of NAS, since choosing over subsets of kernels can effectively capture the accuracy-runtime trade-offs similar to their individually trained counterparts. We therefore conjecture that our efficient **superkernel**-based design search can be flexibly adapted and benefit the guided search space exploration in other RL-based NAS methods. Beyond the NAS literature, our finding is closely related to Slimmable networks [135]. SlimmableNets limit however their analysis across the channel dimension, and our work is the first to study trade-offs across the NAS kernel dimension.

5.3.5 *Single-Path* NAS as feature extractor: COCO object detection

Last, we assess the performance of *Single-Path* NAS as a feature extractor for object detection. and we use our network as a drop-in replacement for the backbone featurizer in the Mask-RCNN model [45]. Similarly, we compare with other backbones networks based on models from earlier mobile NAS methods. We train our model on the COCO dataset [75]. We use the open-source implementation of TPU-trained Mask-RCNN[4]. The models are trained on TPUs with batch size of 64. We train the different models on COCO `train2017` and we evaluate them on COCO `val2017`. We summarize the results on COCO validation set in Table 5.3.

In particular, we observe that our designed DNN achieves higher average-precision (AP) on the validation set compared using backbones from previous mobile AutoML methods. We note that the Mask-RCNN head is less hardware efficient compared to MobileNet-like alternatives such as SSDLite [102]. Nonetheless, the focus of this analysis is to assess the NAS designs as feature extractors while assuming

[4] https://cloud.google.com/tpu/docs/tutorials/mask-rcnn

Table 5.3: COCO Object Detection Performance

Method	AP	AP_S	AP_M	AP_L
MobileNet-V2 + Mask-RCNN	30.47	16.49	32.33	41.14
MnasNet + Mask-RCNN	32.47	17.74	34.45	43.88
ProxylessNAS + Mask-RCNN	32.93	17.76	34.86	44.43
Single-Path NAS + Mask-RCNN (Proposed)	**33.03**	**17.82**	**35.48**	**44.76**

the head design fixed. In fact, recent NAS work has extended the NAS search directly to the backbone [15] and the detection head design [35]. Such searchable components can be flexibly incorporated into our *Single-Path* NAS flow, whose exploration is an interesting direction for future work.

5.4 Discussion

In this Chapter, we propose *Single-Path* NAS, a NAS method that reduces the search cost for designing hardware-efficient DNNs to **less than 4 hours**. The key idea is to revisit the *one-shot* **supernet** design space with a novel *single-path* view, by formulating the NAS problem as *finding which subset of kernel weights to use* in each DNN layer. *Single-Path* NAS reduces the search cost of hardware-efficient NAS down to only **8 epochs** (30 TPU-hours), which is up to **5,000× faster** compared to prior work. While we used a differentiable NAS formulation, our novel design space encoding can be flexibly incorporated into other NAS methodologies. Hence, an interesting line of future work is to combine the efficiency of our *one-shot* design space with RL- or EA-based NAS search strategies.

Another insightful research direction is to investigate how the performance of *single-path one-shot* formulations can be improved further. In the next Chapter, we delve into the various factors that affect the accuracy of the identified DNN design: search space, differentiable solver formulation, and solver parameterization. Our exploration yields state-of-the-art Mobile AutoML results.

Chapter 6

Exploring the Neural Architecture Search Space

In this Chapter, we investigate the key components of a state-of-the-art *one-shot* NAS methodology: the search space, the differentiable solver formulation, and the solver parameterization. Our goal is to assess how each one of these aspects affects the NAS performance in terms of the accuracy of the identified DNN. Our findings push the state-of-the-art performance of Mobile AutoML methods further, while maintaining the overall search cost down to a few hours.

6.1 Chapter overview

The AutoML community has motivated the significance of understanding the properties of NAS solvers and their limitations [54, 1, 83]. Nonetheless, a significant number of existing NAS methods are mainly driven by strong empirical results [97, 137, 12]. This leaves a plethora of interesting questions to investigate: "*how the different NAS formulations, e.g., the encoding of NAS choices across multiple paths or a single path, affect the overall performance of differentiable NAS*." Besides an inter-approach comparison, prior work on mobile NAS [128, 12] lacks a detailed intra-level analysis on the statistics of differentiable NAS. That is, "*by how much the quality of the DNN design varies across multiple runs of the same NAS search?*"

Developing some understanding of the factors that affect the performance of a NAS solver is paramount to enhancing AutoML methodologies [1]. Such endeavor necessitates a comprehensive set of exploratory experiments with the intention of analyzing the properties of a NAS system. A number of existing methods share a similar goal. For instance, the properties of weight sharing within the *one-shot* supermodel are studied in [1]. Similarly, Stochastic NAS [132] investigates the entropy of DNN distributions, while the analysis in [70] shows that random search is a good baseline compared to complex NAS methodologies.

However, existing analyses are limited to the cell-based design based on DARTS [79] and the properties are studied against *proxy* datasets, such as CIFAR-10. This could be attributed to the high computational

burden that is intrinsic to NAS exploration. In this Chapter, we exploit the efficiency of novel *single-path one-shot* formulations (as introduced in Chapter 5) to delve into the key aspects of a NAS solver. Specifically, we identify the following three factors to investigate: (i) the differentiable solver formulation, (ii) the solver parameterization, and (iii) the search space.

First, in terms of the solver formulation, we aim to quantitatively investigate various *single-* and *multi-path* methods with different modeling approximations to encode/relax NAS decisions (*e.g.*, *sigmoid*, *softmax*, or *STE* functions). Second, in terms of solver parameterization, we note that existing methods approximate Pareto optimal solutions by a customized weighted objective function based on a trade-off parameter λ. Nonetheless, this value is manually picked. For instance, MnasNet employs an empirical rule based on "prior" runtime-accuracy trade-off knowledge [122], while FBNet [128] and ProxylessNAS [12] do not provide details on the λ value used or how it was picked. This gives rise to the following question: "*What is the value of the trade-off parameter λ that yields Pareto optimal solutions around a target latency?*"

Last, we strive to enhance the NAS search space in order to further improve the DNN accuracy *vs*. hardware efficiency trade-off. In particular, we investigate the addition of a Squeeze-and-Excitation [52] (SE) path in the MobileNet-based search space. Prior work shows that the SE path can improve the overall accuracy [49]. This finding has been adapted by recent RL-based designs [122, 123]. Nonetheless, the exploration is limited to a binary decision of using SE or not. Instead, in our work we are the *first* to treat the SE path as fully searchable (*i.e.*, searching over various SE ratios). To this end, we introduce *Single-Path+* NAS which achieves state-of-the-art Mobile AutoML performance [112].

Contributions and Chapter organization

In this Chapter, to the best of our knowledge, we make the following contributions:

1. Our work is the first to formulate the hyperparameter tuning of a differentiable NAS solver as a hyperparameter optimization problem itself, aiming to automatically find the trade-off hyperparameter in differentiable NAS given a target runtime.

2. To the best of our knowledge, we are *first* to treat the Squeeze-and-Excitation [52] (SE) path as a fully searchable operation in the MobileNet-based search space. Our methodology, namely *Single-Path+* NAS, is the first *single-path* NAS approach with SE paths.

3. *Single-Path+* NAS [112] achieves a new state-of-the-art: 75.62% top-1 accuracy on ImageNet with $\sim 80ms$ latency on a Pixel 1, *i.e.*, a $+0.42\%$ improvement over the previously best hardware-aware

NAS [122] and manually-designed [49] DNNs in similar latency settings, while maintaining the efficiency of *single-path one-shot* formulations (*i.e.*, 2.45 hours on TPU-v3, 24 TPU-hours).

The remainder of this Chapter is organized as follows: Sections 6.2 and 6.3 investigate the performance of *one-shot* NAS under various solver formulations and parameterization schemes, respectively. Section 6.4 extends the Mobile NAS search space and discusses the state-of-the-art results for image classification performance. Last, Section 6.5 provides the discussion.

6.2 Investigating one-shot NAS formulations

To comprehensively investigate various single- and multi-path methods, we consider the following differentiable NAS formulations:

1. Multi-path with sigmoid: This implementation solves the bilevel, multi-path formulation of Equation 2.3. To this end, we implement a vanilla differentiable multi-path NAS solver [12]. While our implementation replicates prior work's methodology [128], we adjust the multi-path solver to the aggressive few-epochs schedule used in [122, 113]. This allows us to assess whether existing multi-path methods can reach a high-performing DNN within the same number of epochs as *Single-Path NAS*. Specifically, we set the number of total steps to eight epochs and we update the warm-up and learning rate schedules accordingly. We slim down the **multi-path supernet** by a width-multiplier factor of 0.5 (recent NAS work also employs such search on a scaled-down model [123]). Similar to [128], we generate a proxy dataset (*i.e.*, subset of ImageNet with 100 classes) to search on. We deploy our implementation on cloud TPUs.

2. Single-path with sigmoid: this is the default implementation reported so far. That is, during search (backpropagation over the supernet) we approximate the indicator functions (*e.g.*, $\mathbb{1}(\left\|\mathbf{w}_{5\times5\setminus3\times3}\right\|^2 > t_{k=5})$) with sigmoid functions $\sigma()$.

3. Single-path with STE [2]: during search we approximate the indicator functions with the straight-through estimator (STE) [134].

4. Single-path with softmax: This implementation is a hybrid between the single-path encoding of the design space and the use of softmax, *i.e.*, we encode the NAS choice of selecting across subsets of the **superkernel** using a softmax function parameterized by τ, *i.e.*, softmax(τ). For instance, we represent the kernel-level decision as:

$$\mathbf{w}_k = \frac{\exp(\tau_{3\times3})}{\sum_j \exp(\tau_j)} \cdot \mathbf{w}_{3\times3} + \frac{\exp(\tau_{5\times5})}{\sum_j \exp(\tau_j)} \cdot (\mathbf{w}_{3\times3} + \mathbf{w}_{5\times5\setminus3\times3}) \tag{6.1}$$

We formulate the *Single-Path* search as a bilevel optimization problem $\min_{\tau} \min_{\mathbf{w}_\tau} \mathcal{L}(\tau, \mathbf{w}_\tau)$, where we alternate the steps for updating the τ parameters and the DNN weights.

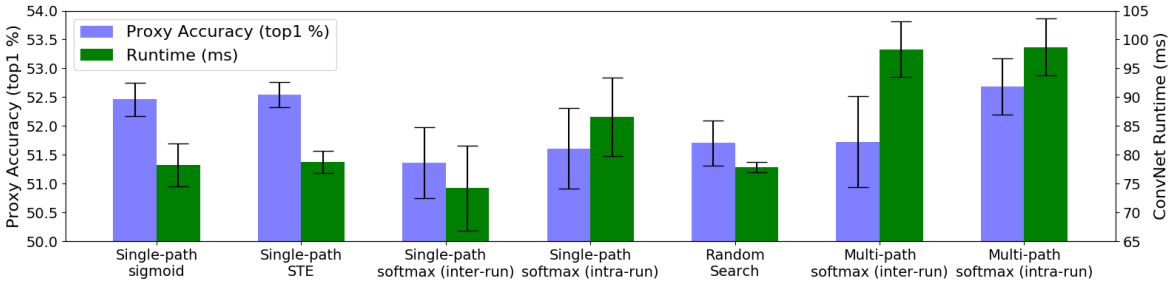

Figure 6.1: *"How the differentiable Mobile NAS formulation assumptions affect the overall performance (accuracy and runtime) of the AutoML-designed DNN?"* Statistics (mean and variance) for the (proxy) accuracy (top 1%) and the runtime of DNNs designed via various formulations across 20 runs; for intra-run statistics, we pick the Pareto optimal DNN out of the 20 samples and we train another 20 DNNs sampled from the softmax distribution.

5. Random search: Parameter-free random search via constrained sampling (by rejection). Samples are limited within the range of interest $\sim 80ms$.

For all the aforementioned methods, we find the λ value that achieves the desired accuracy trade-off $\sim 80ms$ (to tune λ, we use the hyperparameter-tuning scheduler presented in subsection 6.3). We repeat the same NAS search experiment 20 times and we measure the mean and (**inter-**) variance across the 20 runs for both objective terms, *i.e.*, validation accuracy and runtime of the AutoML-designed DNN, denoted as *inter-run*. In addition, to capture the (**intra-**) variance within a single search in softmax-based methods, we pick the best result among the 20 runs, and we train 20 new samples from the softmax distribution (in fact, similar selection is used in [128] where 10 DNNs are sampled and trained to pick the best). We denote the latter variant as *intra-run*. We train each DNN for a few epochs to obtain a representative proxy-accuracy value, following the aggressive training used in MnasNet to study their RL method [122]. We summarize our results in Figure 6.1.

Comparison *vs.* random search: This result is particularly interesting, since there has been recent discussion within the NAS community on whether simple random search could find designs with performance comparable to those of more complex methods [70]. Indeed, we observe that random search performs on par with multi-path cases, which confirms similar observations by recent work [131, 20]. Nonetheless, it is important to note that random search is still inferior compared to *Single-Path NAS* in terms of the (proxy) accuracy around the target latency range $\sim 80ms$.

Furthermore, the nearly-zero search cost of random search is not necessarily representative: to avoid training all random, constraint-satisfying samples, an AutoML practitioner would employ an evaluation of a proxy task, by training each sample for few epochs and by picking the one with highest accuracy. Hence, the actual search cost for random search is not negligible. In fact, the low search cost of our method (8 epochs) is comparable to the number of training epochs during the aforementioned selection process.

Given than *Single-Path NAS* gives DNNs with superior performance than random search at comparable cost, we argue that NAS remains a better AutoML options than random search methods.

Softmax intra-run variance: Next, please notice the variance inherent to all the softmax-based cases. That is, we observe that sampling the softmax of the best NAS search (selected from the 20 NAS repetitions) yields high-variance in terms of both accuracy and runtime. This finding confirms a recent analysis that shows the high entropy in the architecture distribution for cell-based multi-path designs [132].

Different single-path variants: Moreover, we compare our original *Single-Path NAS* (single-path sigmoid) method against its two variants (i) with STE and (ii) with softmax (inter-run). First, once again we notice that the softmax version has higher variance compared to both the sigmoid and the STE versions. For the STE version, while the variance appears smaller than sigmoid, it is important to note that we had to repeat the process multiple times to reach 20 completed searches due to encountered numerical instability issues with STE (exploding gradients). A deeper study on the STE is an interesting direction for future NAS work, similar to recent STE analysis in the context of hardware-aware quantization [134].

Single-Path NAS *vs.* prior work: Last, we highlight the advantage of using our proposed method (single-path sigmoid) instead of existing methods [128, 12] (multi-path softmax, inter-run). We observe that the variance across different *Single-Path NAS* runs is smaller than the variance of softmax-based methods (both inter- and intra-run).

Overall, we observe that multi-path softmax methods sample either low accuracy samples (many layers skipped, which is another issue previously observed [132]) or higher accuracy ones that violate the constraint. We hypothesize that the inferior solutions are due to the fact that the bilevel problem (Equation 2.3) is an intrinsically more complex optimization problem to solve, as also discussed in [79]. That is, it is difficult for the multi-path solver to reach a high quality solution within a few epochs, while our proposed *Single-Path NAS* for the same number of steps is as costly as training a compact model.

Besides the optimization complexity, one would argue that the performance of multi-path methods is decided by several hyperparameters. Indeed, we extensively experimented with numerous settings by varying the number of epochs between the interleaved steps (NAS *vs.* DNN weights updates), the learning rates for each update step, the batch size, the Gumbel-softmax parameters [128], to name a few. Given that running each solver parameterization is expensive (hundreds of epochs), this highlights another limitation related to the tuning cost for all the hyperparameters involved, making our proposed method even more appealing to use. Next, we investigate how the parameterization of the NAS solver affects the end result and how we can automate this tuning step.

6.3 Hypertuning the NAS solver

In this Section, we answer this interesting question: *"instead of empirically tuning the NAS trade-off hyper-parameter, can we automatically find it given a target runtime from the hardware engineers?"* To this end, we formulate the tuning of λ (Equation 5.7) as a hyperparameter optimization problem itself. Specifically, we solve for the λ value that maximizes the validation accuracy under the given runtime target R_T. For a representative analysis, we use the weighted objective introduced in [122] that approximates Pareto optimal solutions, hence allowing our approach to traverse the Pareto front while solving for λ. Specifically:

$$\max_{\lambda} \; Acc_{valid}(\lambda|\mathbf{w}, \mathbf{t}_k, \mathbf{t}_e, \mathbf{t}_{se}) \cdot \left[\frac{R(\lambda|\mathbf{w}, \mathbf{t}_k, \mathbf{t}_e, \mathbf{t}_{se})}{R_T}\right]^w, \text{ with } w = \begin{cases} 0, & \text{if } R(\lambda|\mathbf{w}, \mathbf{t}_k, \mathbf{t}_e, \mathbf{t}_{se}) \leq R_T \\ -1, & \text{otherwise} \end{cases} \tag{6.2}$$

We would like to stress here that each evaluation of Equation 6.2 corresponds to new NAS search. Therefore, solving this hyperparameter optimization problem would be impractical with previous NAS methods where each function evaluation would cost hundreds of hours. Instead, we exploit the efficiency of Single-Path NAS and we investigate various *black-box* hyperparameter optimization techniques. Specifically, we consider the following methods:

1. Bayesian optimization [103]: Vanilla Bayesian optimization, as implemented in the `Dragonfly` tool [62], available online[1]. The method fits a Gaussian process (GP) [95] (probabilistic model) to the objective (Equation 6.2) by points sampled across the hyperparameter λ.

2. Multi-fidelity optimization [60]: Enhanced Bayesian optimization method where the GP fits both the hyperparameter space (λ values) and the *fidelity* (budget) space. The intuition is that low-fidelity evaluations could offer a good view of the function manifold at lower cost. We use discrete budget choices from two up to eight (the default maximum in the vanilla case) as multiple fidelities. We use the multi-fidelity method from `Dragonfly` [62] which, for each new sample to evaluate, it suggests the new λ value and the sample budget (epochs).

3. Random search [3]: Parameter-free random search that randomly samples λ values.

We extend our AutoML framework to support this hyperparameter optimization. Our implementation automates the process of launching multiple (sequential or parallel) runs on cloud TPUs and calls the *black-box optimization* solver that suggests the next λ value to evaluate. Our goal is to find the trade-off λ value that yields Pareto-optimal designs around the target runtime level $R_T = 80ms$. We run each solver for five runs with a total budget of 400 epochs and we track the current-best objective value. In Figures 6.2 (right) and Figure 6.2 (left), we report the objective value and the distance from the target runtime, respectively, where we plot the average-best and the variance across the five runs.

[1] https://github.com/dragonfly/dragonfly/

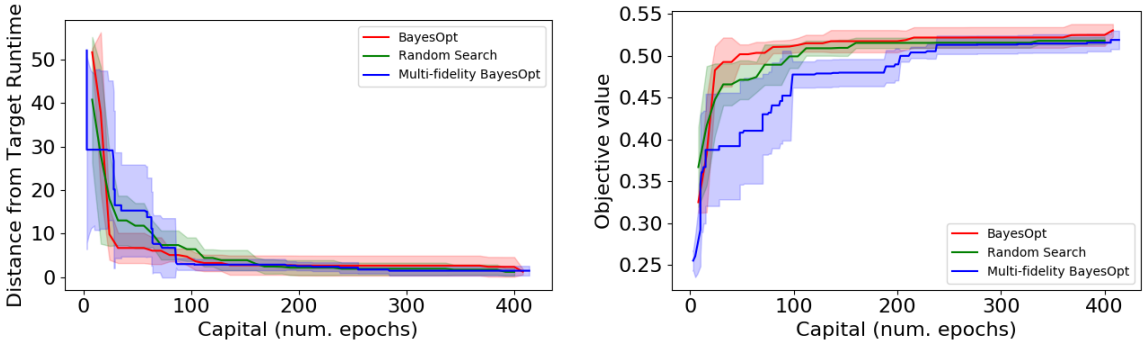

Figure 6.2: Progress of various hyperparameter optimization solvers with respect to the distance from the target latency (left) and the overall reward (right).

Vanilla *vs.* multi-fidelity Bayesian optimization: we observe that vanilla Bayesian optimization outperforms the multi-fidelity counterpart by reaching the near-optimal region faster and by converging to a higher-reward final solution. This is an interesting finding, since prior work shows that, for other hyperparameter settings (*e.g.*, learning rate) multi-fidelity enhances the optimization process.

To fully investigate why this occurs, we employ grid search across budget epochs (from two to eight) and different λ values, and we plot the objective value (Equation 6.2) of the NAS search result in Figure 6.3. The result explains the suboptimality of the multi-fidelity case, since we can observe that the main assumption that "low-cost samples give a representative view of the space" does not fully hold. In particular, as highlighted in the Figure, we observe that initially promising λ values (brighter objective values obtained after four or five epochs, middle right) become suboptimal (darker at eight epochs, bottom right).

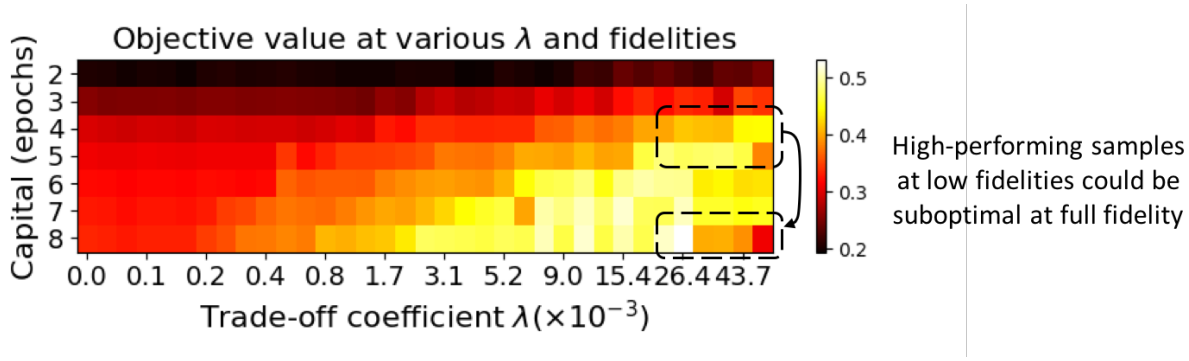

Figure 6.3: Visualizing the objective value (Equation 6.2) across multiple fidelities (y-axis) and hyperparameter values (x-axis) via grid search. Interestingly, low-cost function evaluations (middle, right) that reach the Pareto point around the target latency faster, tend to "overshoot" beyond this point towards over-constrained, suboptimal designs (bottom, right).

From a NAS design standpoint, the larger values λ penalize the runtime term more so they approach the Pareto point around the target latency faster, but they tend to "overshoot" beyond this point towards over-constrained designs. We find this result interesting, since we postulate that other *black-box* optimization

techniques that rely on low-cost (early) approximation (*e.g.*, hyperband [71]) would encounter the same issue. Studying this hyperparameter optimization problem is an interesting research direction for future work currently under-explored.

Comparison *vs.* random search: We find that random search, while never outperforming the Bayesian optimization result, has a relatively good performance at tuning the λ hyperparameter. Interestingly, recent work shares similar observation when tuning NAS scaling hyperparameters via grid search [123]. We hope that our analysis would further foster exploration towards this direction.

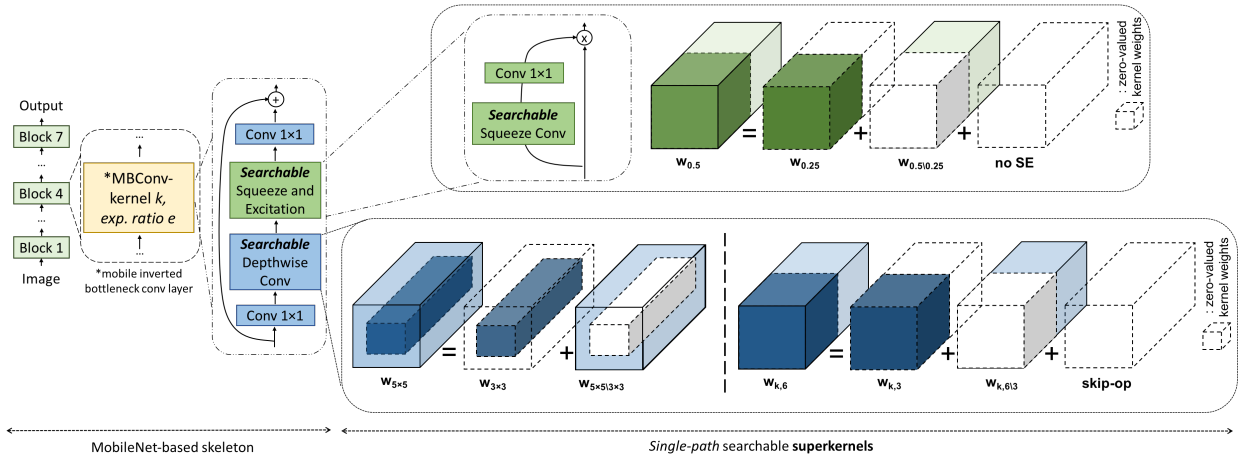

Figure 6.4: *Single-Path+* search space [112]: we enhance the MobileNet-based space with fully searchable Squeeze-and-Excitation [52] (SE) paths. Our method searches over the weights of both the depthwise searchable **superkernel** (*i.e.*, kernel size and expansion ratio values) and the searchable *squeeze* **superkernel** (*i.e.*, SE ratio value). We show that this search space further improves the accuracy-runtime trade-off.

6.4 *Single-Path+*: enhancing the *one-shot* NAS search space

We enhance the Mobile AutoML design space, as is shown in Figure 6.4. In particular, each mobile inverted bottleneck convolution MBConv [102] micro-architecture is now also augmented with a Squeeze-and-Excitation (SE) block [52]. That is, besides the kernel size and the expansion ration, each MBConv-$k \times k$-e-se layer is also parameterized by the Squeeze-and-Excitation [52] ratio se, *i.e.*, the ratio between the number of channels in the intermediate convolution and the input of the Squeeze-and-Excitation path.

Searching for SE ratio: Next, we extend the **superkernel**-based definition to encode the NAS decision related to the Squeeze-and-Excitation [52] (SE) ratio se. In particular, we observe that the expansion ratio decision (Equation 5.3) corresponds effectively to searching over the total number of channels of a convolution kernel. As shown in Figure 6.5, we replace the convolution kernel of the *squeeze* convolution of the SE path with a searchable superkernel, where the largest number of channels corresponds to the

largest candidate *se* value, *i.e.*, $se = 0.5$. By following an intuition similar to Equation 5.3, we observe that "zero-ing out" the second half of the *squeeze* convolution corresponds to using $se = 0.25$, while "zero-ing out" the entire kernel corresponds to not using a SE path ($se = 0$). We therefore write:

$$\mathbf{w}_{se} = \mathbb{1}(\|\mathbf{w}_{0.25}\|^2 > t_{se=0.25}) \cdot (\mathbf{w}_{0.25} + \mathbb{1}(\left\|\mathbf{w}_{0.5\setminus 0.25}\right\|^2 > t_{se=0.5}) \cdot \mathbf{w}_{0.5\setminus 0.25}) \tag{6.3}$$

Overall, we now have two searchable **superkernels**, the original superkernel across the main MBConv path, and the superkernel across the SE path. For input $\mathbf{x}$, the output of the i-th MBConv becomes:

$$o^i(\mathbf{x}) = \text{conv}(\mathbf{x}, \mathbf{w}^i | t^i_{k=5}, t^i_{e=6}, t^i_{e=3}, t^i_{se=0.5}, t^i_{se=0.25}) \tag{6.4}$$

Last, we need to properly incorporate this NAS decision into the runtime term. To this end, we capture the effect that the SE path has on the runtime. For notation consistency, we denote the total runtime of the i-th MBConv layer with kernel size k, expansion ratio e, and SE ratios 0.25 or 0.5 as $R^i_{k\times k,e,se=0.25}$ and $R^i_{k\times k,e,se=0.5}$, respectively. Similarly, we denote the runtime of the MBConv layer without a SE path as $R^i_{k\times k,e,se=0}$. For notation clarity, let us denote the relative increase in runtime due to the addition of the SE path compared to the runtime without the SE path as scaling factor:

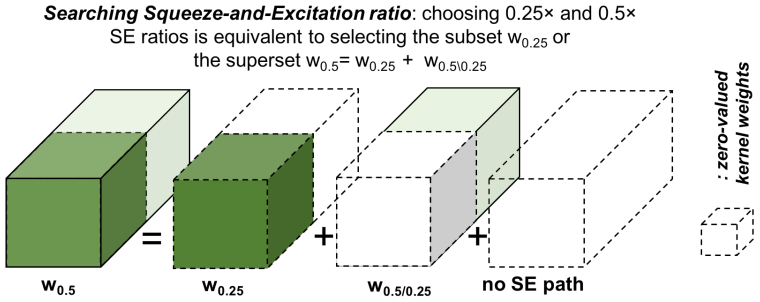

Figure 6.5: Encoding NAS decisions into the **squeeze superkernel**: We formulate all candidate Squeeze-and-Excitation (SE) path types (*i.e.*, SE ratio values) directly into the **searchable superkernel**.

$$s^i_{k,e,0.25} = R^i_{k\times k,e,se=0.25} / R^i_{k\times k,e,se=0} \tag{6.5}$$

Based on our detailed runtime analysis presented in our results, we make two observations: (i) due to the relatively smaller size of the *squeeze* convolution across the SE path compared to the $k \times k$ convolution of the main path, the difference in the relative runtime increase from using either SE ratios is negligible, *i.e.*, $s^i_{k,e,0.25} \approx s^i_{k,e,0.5}$. Next, (ii) the relative ratio of the runtimes with and without using the SE path differs for each type of the main MBConv path. To this end, we express the overall runtime scaling as function of the kernel and the expansion ratio choices:

$$s^i_{k,e=6,0.25} = \mathbb{1}(\left\|\mathbf{w}_{5\times 5\setminus 3\times 3}\right\|^2 > t_{k=5}) \cdot s^i_{k=5,e=6,0.25} + (1 - \mathbb{1}(\left\|\mathbf{w}_{5\times 5\setminus 3\times 3}\right\|^2 > t_{k=5}) \cdot s^i_{k=3,e=6,0.25} \tag{6.6}$$

$$s^i_{k,e=3,0.25} = \mathbb{1}(\left\|\mathbf{w}_{5\times 5\setminus 3\times 3}\right\|^2 > t_{k=5}) \cdot s^i_{k=5,e=3,0.25} + (1 - \mathbb{1}(\left\|\mathbf{w}_{5\times 5\setminus 3\times 3}\right\|^2 > t_{k=5}) \cdot s^i_{k=3,e=3,0.25} \tag{6.7}$$

Hence, overall we have:

$$R^i = \left(1-\mathbb{1}(\left\|\mathbf{w}_{0.5\backslash 0.25}\right\|^2 > t_{se=0.25})\right)\cdot R^i_{k,e} + \mathbb{1}(\left\|\mathbf{w}_{0.5\backslash 0.25}\right\|^2 > t_{se=0.25})\cdot \left\{\mathbb{1}(\left\|\mathbf{w}_{k,6\backslash 3}\right\|^2 > t_{e=6})\cdot s^i_{k,e=6,0.25} + \left(1-\mathbb{1}(\left\|\mathbf{w}_{k,6\backslash 3}\right\|^2 > t_{e=6})\right)\cdot s^i_{k,e=3,0.25}\right\}\cdot R^i_{k,e} \tag{6.8}$$

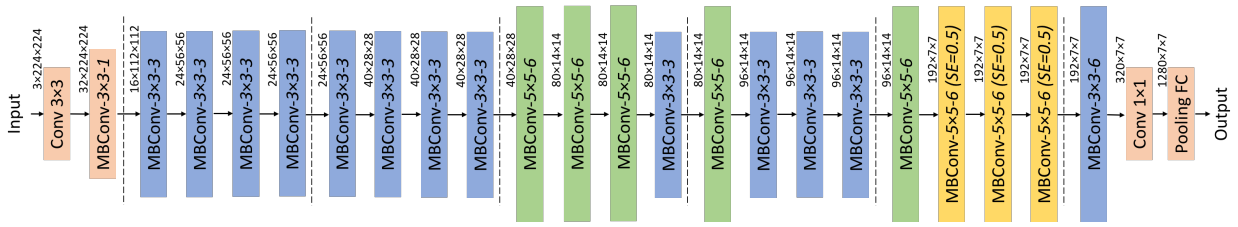

Figure 6.6: Hardware-efficient *Single-Path+* DNN design [112], with top-1 accuracy of **75.62%** on ImageNet and inference time of 81.84*ms* on Pixel 1 phone. Compared to previous DNNs without SE [113] (Figure 5.8), some of the earlier 5×5 MBConvs have been replaced with smaller $3\times 3-3$ MBConvs, and instead *Single-Path+* NAS selects SE paths with SE ratio of $se = 0.5$ in the last layers.

6.4.1 Analyzing the SE-based accuracy-runtime trade-off

Our derived DNN is shown in Figure 6.6. To capture the overhead possibly introduced by the use of the SE path, we report the relative runtime increase per MBConv types for each layer in Figure 6.7. We can make the following observations. First, we observe that the relative increase in the MBConv's runtime (scaling factor $s_{k,e,0.25}$ in Equation 6.8) is closer to 1.0 for the last 4 layers. This is to be expected, since the *squeeze* 1×1 convolution is performed on input feature maps with reduced spatial dimensions.

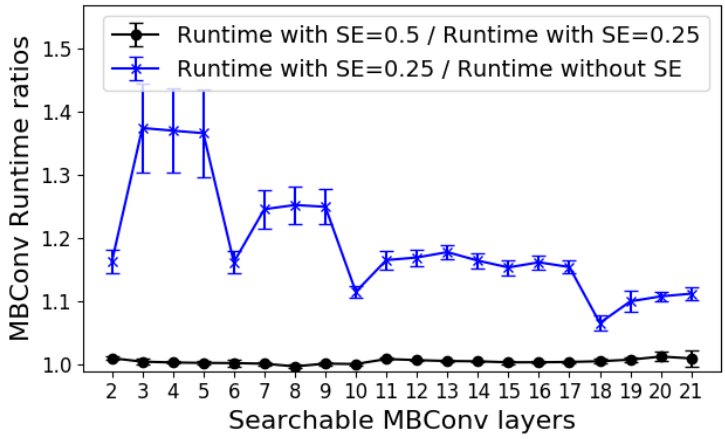

Figure 6.7: Runtime profiling shows that SE ratios larger than 0.25 provide a better accuracy-runtime trade-off, since the *squeeze* step is enhanced with more channels with negligible runtime overhead ($s^i_{k,e,0.25} \approx s^i_{k,e,0.5}$), especially for the deeper layers (MBConv 18-21).

Indeed, we observe that the *Single-Path NAS* appends SE paths in these last layers. More importantly, we observe that the difference in the relative runtime increase from using either SE ratios of 0.25 or 0.5 is negligible, *i.e.*, $s^i_{k,e,0.25} \approx s^i_{k,e,0.5}$. This is important in the context of NAS decision since prior work only

Table 6.1: *Single-Path+* NAS, enhanced with fully searchable Squeeze-and-Excitation [52] (SE) paths, further pushes the state-of-the-art accuracy (%) on ImageNet for the targeted mobile latency setting ($\approx 80ms$ on Pixel 1), currently outperforming both manually- and NAS-designed DNNs that also consider SE [49, 122]. † For MobileNetV3, we report the version that matches the MnasNet space backbone, since some additional manual enhancements in the network head are directly applicable to all other DNNs below.

Method	Top-1 Acc (%)	Top-5 Acc (%)	Mobile Runtime (ms)	Search Cost (epochs)
MobileNetV2 [102]	72.00	91.00	75.00	-
MobileNetV3 [49]	75.20	–	78 †	
MnasNet-B1 [122]	74.00	91.80	76.00	40,000
MnasNet-A1 [122]	75.20	92.50	78.00	
ProxylessNAS-R [12]	74.60	92.20	78.00	200
Single-Path NAS [113]	74.96	92.21	79.48	8
Single-Path+ NAS (**proposed**)	**75.62**	**92.61**	81.84	**8 (2.45 hours)**

searches over the binary decision of using $se = 0.25$ or not, without searching for the se value. Indeed, *Single-Path+* NAS uses a ratio of $se = 0.5$ for all the SE layers.

6.4.2 State-of-the-art Mobile AutoML results

To enable a representative evaluation, we train the DNN architecture identified by our SE-enhanced AutoML method following the same training setup as in prior work [113, 122]. Table 6.1 shows that *Single-Path+* NAS achieves top-1 accuracy of **75.62%** with $\sim 80ms$ latency on a Pixel 1. This is a new state-of-the-art ImageNet accuracy among hardware-efficient NAS methods, *i.e.*, a $+0.42\%$ improvement compared to the previously best hardware-aware NAS model (MnasNet-A1 [122]) and DNNs that combine both AutoML and manual-design expertise (MobileNetV3 [49]). Interestingly, both these methods search over a similar design space, augmented with an SE path.

Moreover, our proposed approach fully maintains the search cost efficiency of a *single-path one-shot* formulation, hence being orders of magnitude faster compared to all previous hardware-efficient NAS methods. That is, we search for $\sim 10k$ steps (8 epochs with a batch size of 1024), which corresponds to total wall-clock time of **2.45 hours** on a TPUv3-8 (*i.e.*, 24 TPU-hours). Our method has a total cost of eight epochs, which is **5,000**$\times$ faster than MnasNet and $25\times$ faster than ProxylessNAS.

Last, compared to the *Single-Path* NAS [113] design, we observe that some of the earlier MBConv types with either 5×5 kernels or expansion ration 6, have been replaced with smaller $3 \times 3 - 3$ MBConvs, and instead the *Single-Path+* NAS flow selects SE paths with SE ratio of $se = 0.5$ in the last few layers. As the obtained top-1 accuracy of the fully-trained DNN attests, the use of a searchable SE improves the accuracy-runtime trade-off of mobile DNNs.

6.5 Discussion

In this Chapter, we delve into the key components of a *one-shot* NAS solver: the search space, the differentiable solver formulation, and the solver parameterization. We assess how various implementation choices affect the performance in terms of the identified DNN. Moreover, we exploit the search-cost efficiency of *single-path* NAS to explore novel dimensions in the search space. Our enhanced NAS methodology *Single-Path+* achieves a new state-of-the-art: 75.62% top-1 accuracy on ImageNet with $\sim 80ms$ latency on a Pixel 1, *i.e.*, a $+0.42\%$ improvement over the previously best AutoML designs. An interesting exploration for future work is to analyze how the accuracy of candidate models (based on *single-path* weight-sharing) correlates to *standalone* proxy accuracy, similar to the analysis in [1] for the *multi-path* case.

Chapter 7

Related Work

Beyond the AutoML methodologies studied so far in this thesis, the field of hardware-efficient deep learning (DL) spans numerous techniques across other AutoML formulations, applications, and design paradigms. In this chapter, we discuss related methods that tackle the problem of efficient DL applications from different viewpoints.

7.1 Modeling the hardware performance of DNNs

Modeling the hardware cost of DNN execution during inference is a critical component across several AutoML methods. Earlier approaches have relied on simplistic proxies such as the DNN parameter and FLOP count to approximate the overall memory consumption and the overall computational cost, respectively [34]. By incorporating Joule-per-operation factors (based on technology node spreadsheets), prior art has proposed counter-based models to approximate energy consumption [99, 105, 43]. Nonetheless, recent work shows that low FLOP or parameter counts do not necessarily translate to hardware efficiency [31, 85]. To address this limitation, recent work on DNN modeling has introduced accurate, regression-based predictive models trained on commercial GPUs [94, 8]. Recent AutoML literature employs similar profiling-based methodologies where the DNN execution on hardware platforms is explicitly profiled and/or modeled [31, 122]. In particular, for mobile AutoML tasks targeting deployment on smartphones, NAS methods model the total runtime as a sum over the per-layer runtimes [12, 128].

Capturing DNN performance on hardware accelerators

The goal of deploying low-power DL models to edge devices, such as IoT nodes, has spurred the development of methodologies that aim to (co-)optimize the design of hardware accelerators [82], hence necessitating the development of accurate hardware simulators that would allow efficient design space

exploration (DSE). Several recently introduced simulators (*e.g.*, MAESTRO [66], SCALE-Sim [101], HERALD [67]) aim to cater to this need, by modeling a wide range of hardware accelerator design choices. Towards more accurate modeling of the accelerator performance, recent work investigates NAS in the presence of manufacturing variability, but these efforts are limited to RTN-induced variability phenomena [58]. Towards this direction, these simulation frameworks provide the foundation where other process variation- [115, 17] or aging-aware models [9, 98, 111, 116, 118, 117, 21] could be flexibly incorporated.

7.2 Towards efficient DNN execution

7.2.1 Adaptive DNNs

Prior art has shown that a large percentage of images in a dataset are easy to classify with a simpler DNN configuration [127]. This insight of dynamically trading off accuracy with energy efficiency can be found in several existing approaches (*e.g.*, conditional [90], scalable [127]). The early efforts to enable energy efficiency were based on "early-exit" conditions placed at each layer of a DNN, aiming at bypassing later stages of a DNN if the classifier has a "confident" prediction in earlier stages. These methods include the scalable-effort classifier [127], the conditional deep learning classifier [90], the distributed neural network [124], the edge-host partitioned neural network [64], and the cascading neural network [69]. At the network-level, Takhirov *et al.* have trained an adaptive classifier [120]. Park *et al.* propose a two-network adaptive design [92], where the decision of which network to process the input data is done by looking into the "confidence score" of the network output. Bolukbasi *et al.* extend the formulation of network-level adaptive systems to multiple networks [7]. Last, other works extend this approach to more tree-like structures for image classification [89], across multiple resolution scales for video object detection [19], or across variant granularity of the image labels [16].

7.2.2 Pruning & quantization

There is an enormous body of work on techniques aiming to reduce the DNN model complexity for efficient hardware execution during inference: *e.g.*, pruning of the network connections [43, 22, 133], quantization of the network weights and activations [26, 27, 25, 28], hardware-efficient implementations of convolution operations [18, 129], to name a few. Recent work shows that these design principles can be directly incorporated into the AutoML search space. In [130], the authors introduce a quantization-aware NAS method to simultaneously search for the arithmetic precision and the DNN architecture. Similarly, Cai *et al*. [10, 11] jointly optimize for arbitrary number of filters and precision while searching the DNN backbone.

That is, such enhancements can be viewed as appending more search dimensions in the AutoML search space, hence providing an interesting line of future work for our methodologies.

7.3 Hardware-aware Bayesian optimization

Prior art has proposed formulations for constrained Bayesian optimization, motivating optimization cases where the constraints can be expressed as known *a priori* [34]; these formulations enable models that can directly capture candidate configurations as valid or invalid [40]. Hernández-Lobato *et al.* developed a general framework for employing Bayesian optimization with unknown constraints or with multiple objective terms [47]. This framework has been successfully used for the co-design of hardware accelerators and DNNs [48, 96], and the design of DNNs under runtime constraints [47]. However, existing methodologies evaluate only MNIST on hardware simulators [48, 96], do not consider power as key design constraint [47], or rely on inaccurate count-based models instead of platform measurements.

7.4 Hardware-aware Neural Architecture Search (NAS)

NAS literature (*standalone* or *one-shot*) has investigated a plethora of solver formulations and search selection strategies, spanning methods based on reinforcement learning (RL) [138, 137], evolutionary algorithms [97], gradient-based methods [79, 93], Bayesian optimization [61, 23], to name a few.

Hardware-aware NAS: Earlier hardware-aware NAS methods focused on maximizing accuracy under FLOPs constraints [132, 136], but low FLOP count does not necessarily translate to hardware efficiency [31]. More recent methods incorporate hardware terms (*e.g.*, runtime, power) into cell-based NAS formulations [31, 51], but cell-based implementations are not hardware friendly [128]. Breaking away from cell-based assumptions in the search space encoding, recent work employs NAS over a generalized MobileNetV2-based design space introduced in [122].

Hardware-aware multi-path differentiable NAS: Recent NAS literature has seen a shift towards one-shot NAS formulations [93, 132]. Gradient-based NAS in particular has gained increased popularity and has achieved state-of-the-art results [79, 84]. One-shot-based methods use an over-parameterized super-model network, where, for each layer, every candidate operation is added as a separate trainable path. Nonetheless, *multi-path* search spaces have an intrinsic limitation: the number of trainable parameters that need to be maintained and updated with gradients during the search grows linearly with respect to the number of different convolutional operations per layer, resulting in memory explosion [1, 12].

To this end, state-of-the-art approaches employ different novel "workaround" solutions. FBNet [128] searches on a "proxy" dataset (*i.e.*, subset of the ImageNet dataset). Despite the decreased search cost

thanks to the reduced number of training images, these approaches do not address the fact that the entire supermodel needs to be maintained in memory during search, hence the efficiency is limited due to inevitable use of smaller batch sizes. ProxylessNAS [12] has employed a memory-wise one-shot model scheme, where only a set of paths is updated during the search. However, such implementation-wise improvements do not address a second key suboptimality of one-shot approaches, *i.e.*, the fact that separate gradient steps are needed to update the weights and the architectural decisions interchangeably [79]. Although the number of trainable parameters, with respect to the memory cost, is kept to the same level at any step, the way that *multi-path*-based methods traverse the design space remains inefficient.

Hardware-aware single-path differentiable NAS: While concurrent methods consider relaxed convolution formulations based on insights similar to our work [104, 53, 42], they either use design spaces and objectives that have been shown to be hardware inefficient (*e.g.*, cell-based space, FLOP count), or they optimize over a subset of our design space. In our work, we optimize over multiple searchable kernels per layer and we simultaneously search across several NAS decisions, *i.e.*, kernel sizes, channels dimensions, expansion ratio, or Squeeze-and-Excitation [52] ratio dimensions.

Chapter 8

Conclusion

In this thesis, we have demonstrated that AutoML methods can be both *hardware aware* and search-cost *efficient*: our proposed methodologies can design hardware-efficient DNN architectures that achieve state-of-the-art DL performance, in only a few hours. In this chapter, we summarize the key results of our work and we discuss several interesting directions for future research that arise of our endeavor.

8.1 Key thesis results

8.1.1 Enhancing Bayesian optimization with hardware constraint-awareness

Vanilla BO treats the hardware cost as a *low-cost*, *a priori*-known constraint: with several point evaluations, the BO solver models the likelihood that a point in the design space satisfies the constraints or not. **Key insight**: we conjugate that BO methods could exploit a more accurate understanding of the underlying hardware cost via accurate models which are trained offline. We train predictive models to capture the power and memory consumption of DNNs running on GPUs, with an overall prediction accuracy of 93%. Based on our key insight, we develop *HyperPower*, a BO-based method where the hardware-cost terms are explicitly incorporated into the formulation, allowing for the solver to traverse the design space in a constraint "complying" manner [109]. We show that *HyperPower* reaches the near-optimal region up to **3.5**$\times$ faster compared to vanilla constrained BO methods.

8.1.2 AutoML for designing adaptive DNNs

The literature on adaptive DNNs has mainly focused on learning where each input image should be classified among the DNNs, hence only optimizing with respect to the selection scheme. That is, prior work treats each DNN as a *blackbox* (*i.e.*, pre-trained off-the-shelf DNN). **Key insight**: we demonstrate

that the hardware efficiency of adaptive DNNs can be greatly improved if the DNN architectures are optimized jointly with the network selection scheme. Our work is the first to formulate the design of adaptive DNNs as an AutoML problem under various energy, accuracy, and communication constraints. We identify designs that outperform existing resource-constrained adaptive DNNs by up to 6× in terms of minimum energy per image and by up to 31.13% in terms of accuracy improvement, when tested on a commercial NVIDIA mobile board and the CIFAR-10 dataset.

8.1.3 *Single-path one-shot* NAS

Despite the strong empirical results in reducing the search cost and identifying state-of-the-art DNN designs, existing *one-shot multi-path* NAS formulations exhibit a key suboptimality: by viewing the NAS choices as an operation/path selection problem, each candidate operation is appended as a separate path to the *one-shot* supernet. However, naïvely branching out all paths is inefficient due to the large number of supernet parameters to be maintained and updated during the search. Hence, these techniques remain considerably costly, with an overall computational demand of at least 200 GPU-hours.

To address this suboptimal formulation, we propose *Single-Path* NAS [113]. **Key insight**: our method views the different candidate convolutional operations in NAS as subsets of a **single "superkernel"**, allowing us to solve the NAS problem as finding which subset of kernel weights to use in each DNN layer. We achieve an overall search cost of only **8 epochs** (**3.75 hours** on TPUs-v2), which is up to **5,000× faster** compared to prior work, while outperforming existing Mobile AutoML methodologies in terms of top-1 accuracy on ImageNet with on-par mobile latency ($\approx 80ms$ on a Pixel 1).

8.1.4 State-of-the-art Mobile AutoML performance

While analyzing the factors that affect the performance of a NAS solver is paramount to enhancing AutoML methodologies, existing analyses are limited in their scope due to the high computational burden that is intrinsic to NAS runs. **Key insight**: we exploit the efficiency of our novel *single-path one-shot* formulation and we delve into the key aspects of a NAS solver. Specifically, we study how NAS implementation choices such as the solver formulation or parameterization affect the search result. Moreover, we enhance the NAS search space by treating the Squeeze-and-Excitation [52] (SE) path as a fully searchable operation. Our enhanced *Single-Path*+ NAS [112] achieves a new state-of-the-art: 75.62% top-1 accuracy on ImageNet with $\sim 80ms$ latency on a Pixel 1, *i.e.*, a $+0.42\%$ improvement over the previously best hardware-aware NAS [122] and manually-designed [49] DNNs in similar latency settings, while maintaining the efficiency of *single-path one-shot* formulations (*i.e.*, 2.45 hours on TPU-v3, 24 TPU-hours).

8.2 Future work

AutoML constitutes a research topic of paramount importance, being viewed as a key factor towards the *democratization* of DL [54]: the user would simply provide the data and AutoML would offer state-of-the-art DNN solutions without the need for DL experts. As of this writing, we are witnessing a proliferation of AutoML approaches, as demonstrated by the exponential increase in the number of AutoML papers [76]. The research community has highlighted that, despite the steady improvement in AutoML methods, the comprehensiveness of evaluations in the field still lags behind compared to other areas in DL, ML, and optimization [76]. Therefore, future work in AutoML, and especially in hardware-aware NAS, can explore numerous interesting directions.

8.2.1 Jointly exploring the underlying hardware architecture space

While recent hardware-aware AutoML has achieved state-of-the-art performance for on-device DL applications, such effectiveness is bound by the underlying hardware resources [10]. The need to bring low-power DL models to edge devices, such as IoT nodes, has spurred the development of NAS methodologies that aim to *jointly optimize* the hardware accelerator design as well as the DNN [82]. *Co-design* NAS approaches present several opportunities and challenges for future work. That is, the extra degree of freedom from the underlying architecture design is expected to make the search complexity larger. To this end, recent work aims to adapt existing *standalone* and multi-path NAS approaches to enable efficient *co-design* [82, 10]. Therefore, a straight-line future direction is to extend our *single-path one-shot* formulation to jointly optimize the underlying accelerator design.

8.2.2 NAS beyond image classification

The NAS literature has mainly focused on image classification. Hence, an interesting direction for future work is to extend AutoML to other DL applications. As of this writing, we note some initial steps towards this direction: NAS for object detection (*e.g.*, designing feature pyramid networks [35] and/or backbones [15]), sequence-to-sequence tasks [108], semantic segmentation [77], to name a few. However, these early "post-ImageNet" techniques are heavily biased to the knowledge from manual engineering: *i.e.*, the search spaces and formulations are defined around previously best hand-tuned architectures (*e.g.*, the evolutionary algorithm seeded around Transformer-like modules [108]). As also noted in [54], these *a priori* assumptions impose intrinsic limitations to the power of NAS methods. Thus, another direction of interest is to explore more general and flexible search spaces.

8.2.3 AutoML in distributed training settings: federated learning

Last, we note that current AutoML approaches assume a conventional, well-defined training setting, *i.e.*, they train and evaluate candidate models on training and validation sets, respectively, that assume to have all datapoints readily available. Such assumptions give rise to an interesting direction for future work towards extending AutoML to novel model training paradigms, such as *federated learning* [72]. In *federated learning*, the goal is to train DL models over a dataset of points that reside on remote distributed devices. Each device may generate data that vary in terms of the underlying data distribution and the number of datapoints, hence necessitating a departure from conventional training approaches [13, 41]. Consequently, extending AutoML methods to a distributed training setting while properly accounting for the inherent *statistical heterogeneity* over heterogeneous and massive networks poses numerous challenges and opportunities for future work.

Bibliography

[1] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558, 2018. 1, 10, 32, 44, 55, 58

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 46

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012. 5, 19, 49

[4] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. Citeseer, 2013. 12

[5] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011. 1, 7

[6] Kartikeya Bhardwaj, Ching-Yi Lin, Anderson Sartor, and Radu Marculescu. Memory-and communication-aware model compression for distributed deep learning inference on iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019. 25

[7] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536, 2017. 21, 23, 26, 28, 29, 57

[8] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. *arXiv preprint arXiv:1710.05420*, 2017. 2, 15, 37, 56

[9] Ermao Cai, Dimitrios Stamoulis, and Diana Marculescu. Exploring aging deceleration in finfet-based multi-core systems. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 111. ACM, 2016. 57

[10] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Kuan Wang, Tianzhe Wang, Ligeng Zhu, and Song Han. Automl for architecting efficient and specialized neural networks. *IEEE Micro*, 2019. 57, 62

[11] Han Cai, Tianzhe Wang, Zhanghao Wu, Kuan Wang, Ji Lin, and Song Han. On-device image classification with proxyless neural architecture search and quantization-aware fine-tuning. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 0–0, 2019. 57

[12] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. ix, 2, 3, 9, 10, 32, 35, 37, 39, 40, 44, 45, 46, 48, 54, 56, 58, 59

[13] Sebastian Caldas, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018. 63

[14] Bo Chen and Jeffrey Gilbert. Introducing the CVPR 2018 On-Device Visual Intelligence Challenge. https://ai.googleblog.com/2018/04/introducing-cvpr-2018-on-device-visual.html, 2018. 2

[15] Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Chunhong Pan, and Jian Sun. Detnas: Neural architecture search on object detection. *arXiv preprint arXiv:1903.10979*, 2019. 43, 62

[16] Zhuo Chen, Ruizhou Ding, Ting-Wu Chin, and Diana Marculescu. Understanding the impact of label granularity on cnn-based image classification. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 895–904. IEEE, 2018. 57

[17] Zhuo Chen, Dimitrios Stamoulis, and Diana Marculescu. Profit: priority and power/performance optimization for many-core systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017. 57

[18] Zhuo Chen, Jiyuan Zhang, Ruizhou Ding, and Diana Marculescu. Vip: Virtual pooling for accelerating cnn-based image classification and object detection. *arXiv preprint arXiv:1906.07912*, 2019. 57

[19] Ting-Wu Chin, Ruizhou Ding, and Diana Marculescu. Adascale: Towards real-time video object detection using adaptive scaling. *arXiv preprint arXiv:1902.02910*, 2019. 57

[20] Minsu Cho, Mohammadreza Soltani, and Chinmay Hegde. One-shot neural architecture search via compressive sensing. *arXiv preprint arXiv:1906.02869*, 2019. 47

[21] Simone Corbetta, Pieter Weckx, Dimitrios Rodopoulos, Dimitrios Stamoulis, and Francky Catthoor. Time-efficient modeling and simulation of true workload dependency for bti-induced degradation

in processor-level platform specifications. In *Harnessing Performance Variability in Embedded and High-performance Many/Multi-core Platforms*, pages 217–235. Springer, 2019. 57

[22] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *arXiv preprint arXiv:1711.02017*, 2017. 57

[23] 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. *arXiv preprint arXiv:1812.08934*, 2018. ix, 39, 58

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 37

[25] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11408–11417, 2019. 57

[26] Ruizhou Ding, Zeye Liu, RD Shawn Blanton, and Diana Marculescu. Quantized deep neural networks for energy efficient hardware-based inference. In *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*, pages 1–8. IEEE, 2018. 57

[27] Ruizhou Ding, Zeye Liu, Ting-Wu Chin, Diana Marculescu, and RD Blanton. Flightnns: Lightweight quantized deep neural networks for fast and accurate inference. In *2019 Design Automation Conference (DAC)*, 2019. 34, 57

[28] Ruizhou Ding, Zeye Liu, Rongye Shi, Diana Marculescu, and RD Blanton. Lightnn: Filling the gap between conventional deep neural networks and binarized networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 35–40. ACM, 2017. 57

[29] Ruizhou Ding, Dimitrios Stamoulis, Kartikeya Bhardwaj, Diana Marculescu, and Radu Marculescu. Enhancing precipitation models by capturing multivariate and multiscale climate dynamics. In *Proceedings of the 3rd International Workshop on Cyber-Physical Systems for Smart Water Networks*, pages 39–42. ACM, 2017. 4

[30] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. 17, 18, 19

[31] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. *arXiv preprint arXiv:1806.08198*, 2018. 2, 38, 56, 58

[32] Facebook Open Source. Bayesian Optimization in PyTorch. https://www.botorch.org/, 2019. 1, 2, 9

[33] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1436–1445, 2018. 1

[34] Michael A Gelbart, Jasper Snoek, and Ryan P Adams. Bayesian optimization with unknown constraints. *arXiv preprint arXiv:1403.5607*, 2014. 5, 15, 16, 17, 18, 56, 58

[35] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7036–7045, 2019. 1, 43, 62

[36] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017. 1, 6, 12

[37] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*, 2013. 21

[38] Google. Google AutoML Beta. https://cloud.google.com/automl/, 2019. 1, 2

[39] Google CLoud. Cloud Speech-to-Text. https://cloud.google.com/speech-to-text/, 2019. 1

[40] Robert B Gramacy and Herbert KH Lee. Optimization under unknown constraints. *arXiv preprint arXiv:1004.4027*, 2010. 16, 58

[41] Neel Guha, Ameet Talwlkar, and Virginia Smith. One-shot federated learning. *arXiv preprint arXiv:1902.11175*, 2019. 63

[42] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019. 1, 59

[43] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015. 56, 57

[44] Karen Hao. Training a single AI model can emit as much carbon as five cars in their lifetimes. https://www.technologyreview.com/s/613630/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/, 2019. 2, 3

[45] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017. 1, 12, 42

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1, 12

[47] José Miguel Hernández-Lobato, Michael A Gelbart, Ryan P Adams, Matthew W Hoffman, and Zoubin Ghahramani. A general framework for constrained bayesian optimization using information-based search. *The Journal of Machine Learning Research*, 17(1), 2016. 58

[48] José Miguel Hernández-Lobato, Michael A Gelbart, Brandon Reagen, Robert Adolf, Daniel Hernández-Lobato, Paul N Whatmough, David Brooks, Gu-Yeon Wei, and Ryan P Adams. Designing neural network hardware accelerators with decoupled objective evaluations. In *Advances in neural information processing systems - workshop on Bayesian Optimization*, 2016. 13, 15, 58

[49] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*, 2019. ix, 45, 46, 54, 61

[50] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 39

[51] Chi-Hung Hsu, Shu-Huan Chang, Da-Cheng Juan, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Shih-Chieh Chang. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332*, 2018. 2, 58

[52] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018. ix, xii, 3, 10, 11, 45, 51, 54, 59, 61

[53] Andrew Hundt, Varun Jain, and Gregory D Hager. sharpdarts: Faster and more accurate differentiable architecture search. *arXiv preprint arXiv:1903.09900*, 2019. 59

[54] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book. x, 5, 6, 44, 62

[55] Eric Hwang, Peter Vajda, Matt Uyttendaele, and Rahul Nallamothu. Facebook AI Performance Evaluation Platform. https://ai.facebook.com/blog/under-the-hood-portals-smart-camera/, 2019. 1, 2

[56] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014. x, 13, 16, 26

[57] Nan Jiang, Akshay Krishnamurthy, Alekh Agarwal, John Langford, and Robert E Schapire. Contextual decision processes with low bellman rank are pac-learnable. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1704–1713, 2017. 31

[58] Weiwen Jiang, Qiuwen Lou, Zheyu Yan, Lei Yang, Jingtong Hu, Xiaobo Sharon Hu, and Yiyu Shi. Device-circuit-architecture co-exploration for computing-in-memory neural accelerators. *arXiv preprint arXiv:1911.00139*, 2019. 57

[59] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017. 38

[60] Kirthevasan Kandasamy, Gautam Dasarathy, Jeff Schneider, and Barnabás Póczos. Multi-fidelity bayesian optimisation with continuous approximations. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1799–1808. JMLR. org, 2017. 49

[61] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018. 31, 58

[62] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R. Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *arXiv preprint arXiv:1903.06694*, 2019. 1, 49

[63] Alexander Kirillov, Kaiming He, Ross Girshick, Carsten Rother, and Piotr Dollár. Panoptic segmentation. *arXiv preprint arXiv:1801.00868*, 2018. 1

[64] Jong Hwan Ko, Taesik Na, Mohammad Faisal Amir, and Saibal Mukhopadhyay. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. *arXiv preprint arXiv:1802.03835*, 2018. 57

[65] Ron Kohavi and George H John. Automatic parameter selection by minimizing estimated error. In *Machine Learning Proceedings 1995*, pages 304–312. Elsevier, 1995. 5

[66] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 754–768, 2019. 57

[67] Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, and Vikas Chandra. Herald: Optimizing heterogeneous dnn accelerators for edge devices. *arXiv preprint arXiv:1909.07437*, 2019. 57

[68] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998. *URL http://yann.lecun.com/exdb/mnist*, 10:34, 1998. 14

[69] Sam Leroux, Steven Bohez, Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. The cascading neural network: building the internet of smart things. *Knowledge and Information Systems*, 52(3):791–814, 2017. 24, 57

[70] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019. 5, 44, 47

[71] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016. 1, 5, 51

[72] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *arXiv preprint arXiv:1908.07873*, 2019. 63

[73] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017. 1, 12

[74] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017. 1, 12

[75] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014. 42

[76] Marius Lindauer and Frank Hutter. Best practices for scientific research on neural architecture search. *arXiv preprint arXiv:1909.02453*, 2019. 6, 62

[77] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 82–92, 2019. 62

[78] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017. 10

[79] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 1, 2, 6, 7, 9, 10, 30, 31, 32, 35, 44, 48, 58, 59

[80] Stephen Lombardi, Jason Saragih, Tomas Simon, and Yaser Sheikh. Deep appearance models for face rendering. *ACM Transactions on Graphics (TOG)*, 37(4):68, 2018. 1

[81] Stephen Lombardi, Jason Saragih, Tomas Simon, and Yaser Sheikh. Deep Appearance Models for Face Rendering. https://research.fb.com/publications/deep-appearance-models-for-face-rendering/, 2019. 1

[82] Qing Lu, Weiwen Jiang, Xiaowei Xu, Yiyu Shi, and Jingtong Hu. On neural architecture search for resource-constrained hardware platforms. *arXiv preprint arXiv:1911.00105*, 2019. 56, 62

[83] Renqian Luo, Tao Qin, and Enhong Chen. Understanding and improving one-shot neural architecture optimization. *arXiv preprint arXiv:1909.10815*, 2019. 44

[84] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pages 7816–7827, 2018. 32, 58

[85] Diana Marculescu, Dimitrios Stamoulis, and Ermao Cai. Hardware-aware machine learning: modeling and optimization. In *Proceedings of the International Conference on Computer-Aided Design*, page 137. ACM, 2018. 56

[86] Microsoft (GitHub open-source project). NNI (Neural Network Intelligence). https://github.com/microsoft/nni, 2019. 1, 2

[87] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Introducing Ludwig, a Code-Free Deep Learning Toolbox. https://eng.uber.com/introducing-ludwig/, 2019. 1, 9

[88] Giljoo Nam, Chenglei Wu, Min H. Kim, and Yaser Sheikh. Strand-accurate multi-view hair capture. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 1

[89] Priyadarshini Panda, Aayush Ankit, Parami Wijesinghe, and Kaushik Roy. Falcon: Feature driven selective classification for energy-efficient image recognition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(12), 2017. 21, 57

[90] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 475–480. IEEE, 2016. 21, 57

[91] Priyadarshini Panda, Swagath Venkataramani, Abhronil Sengupta, Anand Raghunathan, and Kaushik Roy. Energy-efficient object detection using semantic decomposition. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(9):2673–2677, 2017. 21

[92] Eunhyeok Park, Dongyoung Kim, Soobeom Kim, Yong-Deok Kim, Gunhee Kim, Sungroh Yoon, and Sungjoo Yoo. Big/little deep neural network for ultra low power inference. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 124–132. IEEE Press, 2015. 7, 21, 23, 24, 26, 28, 29, 57

[93] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018. 2, 9, 58

[94] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. 2016. 56

[95] Carl Edward Rasmussen and Christopher KI Williams. *Gaussian processes for machine learning*, volume 1. MIT press Cambridge, 2006. 9, 49

[96] Brandon Reagen, José Miguel Hernández-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. A case for efficient accelerator design space exploration via bayesian optimization. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2017. 13, 58

[97] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018. 1, 2, 6, 31, 44, 58

[98] Dimitrios Rodopoulos, Dimitrios Stamoulis, Grigorios Lyras, Dimitrios Soudris, and Francky Catthoor. Understanding timing impact of bti/rtn with massively threaded atomistic transient simulations. In *2014 IEEE International Conference on IC Design & Technology*, pages 1–4. IEEE, 2014. 57

[99] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Delight: Adding energy dimension to deep neural networks. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 112–117, 2016. 56

[100] Tonmoy Saikia, Yassine Marrakchi, Arber Zela, Frank Hutter, and Thomas Brox. Autodispnet: Improving disparity estimation with automl. *arXiv preprint arXiv:1905.07443*, 2019. 1

[101] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator. *arXiv preprint arXiv:1811.02883*, 2018. 57

[102] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. x, 10, 11, 39, 40, 42, 51, 54

[103] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015. 1, 7, 8, 17, 19, 49

[104] Richard Shin, Charles Packer, and Dawn Song. Differentiable neural network architecture search. *OpenReview*, 2018. 59

[105] Sean C Smithson, Guang Yang, Warren J Gross, and Brett H Meyer. Neural networks designing neural networks: multi-objective hyper-parameter optimization. In *Proceedings of the 35th International Conference on Computer-Aided Design*, pages 1–8, 2016. 17, 19, 56

[106] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012. 1, 5, 7, 12, 17, 18, 19, 25

[107] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180, 2015. 5

[108] David R So, Chen Liang, and Quoc V Le. The evolved transformer. *arXiv preprint arXiv:1901.11117*, 2019. 62

[109] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018. 3, 13, 19, 60

[110] Dimitrios Stamoulis, Ting-Wu Rudy Chin, Anand Krishnan Prakash, Haocheng Fang, Sribhuvan Sajja, Mitchell Bognar, and Diana Marculescu. Designing adaptive neural networks for energy-constrained image classification. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018. 3, 22, 24

[111] Dimitrios Stamoulis, Simone Corbetta, Dimitrios Rodopoulos, Pieter Weckx, Peter Debacker, Brett H Meyer, Ben Kaczer, Praveen Raghavan, Dimitrios Soudris, Francky Catthoor, et al. Capturing true workload dependency of bti-induced degradation in cpu components. In *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pages 373–376. IEEE, 2016. 57

[112] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path mobile automl: Efficient convnet design and nas hyperparameter optimization. *arXiv preprint arXiv:1907.00959*, 2019. xii, xiii, 3, 45, 51, 53, 61

[113] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. *arXiv preprint arXiv:1904.02877*, 2019. xiii, 3, 33, 46, 53, 54, 61

[114] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Device-aware efficient convnet design. *arXiv preprint arXiv:1905.04159*, 2019. 3, 33

[115] Dimitrios Stamoulis and Diana Marculescu. Can we guarantee performance requirements under workload and process variations? In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 308–313. ACM, 2016. 4, 57

[116] Dimitrios Stamoulis, Dimitrios Rodopoulos, Brett H Meyer, Dimitrios Soudris, Francky Catthoor, and Zeljko Zilic. Efficient reliability analysis of processor datapath using atomistic bti variability models. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 57–62. ACM, 2015. 57

[117] Dimitrios Stamoulis, Dimitrios Rodopoulos, Brett H Meyer, Dimitrios Soudris, and Zeljko Zilic. Linear regression techniques for efficient analysis of transistor variability. In *2014 21st IEEE international conference on electronics, circuits and systems (ICECS)*, pages 267–270. IEEE, 2014. 57

[118] Dimitrios Stamoulis, Kostas Tsoumanis, Dimitrios Rodopoulos, Brett H Meyer, Kiamal Pekmestzi, Dimitrios Soudris, and Zeljko Zilic. Efficient variability analysis of arithmetic units using linear regression techniques. *Analog Integrated Circuits and Signal Processing*, 87(2):249–261, 2016. 57

[119] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012, 2013. 1, 12, 19

[120] Zafar Takhirov, Joseph Wang, Venkatesh Saligrama, and Ajay Joshi. Energy-efficient adaptive classifier design for mobile systems. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 52–57. ACM, 2016. 23, 24, 57

[121] Mingxing Tan. MnasNet: Towards Automating the Design of Mobile Machine Learning Models. https://ai.googleblog.com/2018/08/mnasnet-towards-automating-design-of.html, 2018. 1

[122] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018. ix, xii, 2, 7, 10, 36, 37, 38, 39, 40, 45, 46, 47, 49, 54, 56, 58, 61

[123] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019. 45, 46, 51

[124] Surat Teerapittayanon, Bradley McDanel, and HT Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 328–339. IEEE, 2017. 57

[125] Joaquin Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018. 6

[126] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 1

[127] Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. Scalable-effort classifiers for energy-efficient machine learning. In *Proceedings of the 52nd Annual Design Automation Conference*, page 67. ACM, 2015. 21, 23, 57

[128] 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. *arXiv preprint arXiv:1812.03443*, 2018. 2, 3, 9, 10, 31, 32, 35, 37, 39, 40, 44, 45, 46, 47, 48, 56, 58

[129] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9127–9135, 2018. 57

[130] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018. 57

[131] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019. 1, 47

[132] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018. 2, 32, 44, 48, 58

[133] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 2, 57

[134] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. *arXiv preprint arXiv:1903.05662*, 2019. 46, 48

[135] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018. 42

[136] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018. 1, 58

[137] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 6, 44, 58

[138] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017. 1, 2, 6, 10, 31, 58