Improving the automated search of neural network architectures

Improving the automated search of neural network architectures

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät

der Eberhard Karls Universität Tübingen

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

M.Sc. Kevin A. Laube

aus Villingen

Tübingen

2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	16.02.2023
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Andreas Zell
2. Berichterstatter:	Prof. Dr. Andreas Schilling

Abstract

Machine learning is becoming increasingly common in our society, from recommendation systems, audio assistants, and autonomous cars to gadgets like image filters for social media. Many other branches in research and industry are also planning to integrate artificial intelligence in their workflows shortly. However, developing and improving such algorithms for many specific tasks requires corresponding quantities of funding and labor, both of which are often scarce.

In machine learning, automated hyper-parameter optimization techniques are widely used to find suitable training parameters such as learning rates and batch sizes. They do not just reduce the required labor but mostly exceed their human competition in speed and quality. Based on similar concepts, automatically designed neural network architectures achieved state-of-the-art performance on modern tasks for the first time in 2016. The study of such processes, known as *neural architecture search*, quickly gained interest as a possible solution to the shortage of labor and a logical next step in the development of machine learning. This thesis focuses primarily on two aspects of neural architecture search:

Firstly, we systematically analyze and improve a baseline search space for the network latency. Architectures discovered in the revised space design have an equivalent network accuracy but are twice as fast. In a second step, we investigate whether search space designs can be automated as well. The proposed *prune and replace* algorithm can progressively search through and specialize a weakly defined search space, even if it contains vastly more architectures than before. Due to multiple technical optimizations and considerations, the search requires less time than before and can discover better architectures.

Secondly, we study performance predicting methods in different contexts. We conducted a large-scale hardware prediction study for various common predictors and studied in detail how multi-objective architecture search is affected by multiple factors such as predictor quality. We also evaluate a modification to super-networks, a widely used accuracy prediction approach. While the change is currently hard to apply, it results in a consistently improved selection of architectures.

We conclude by presenting UniNAS, a framework built to unify various architecture search concepts and approaches in a single code base. Based on argument trees, experiments can be designed flexibly, in great detail, and even from a graphical user interface.

Kurzfassung

Maschinelles Lernen wird in unserer Gesellschaft immer alltäglicher, von Empfehlungssystemen, Audioassistenten und autonomen Autos bis hin zu Gadgets wie Filtern für Bilder in sozialen Netzwerken. Auch viele andere Branchen in Forschung und Industrie planen, künstliche Intelligenz zeitnah in ihre Arbeitsabläufe zu integrieren. Die Entwicklung und Verbesserung solcher Algorithmen für viele spezifische Aufgaben erfordert jedoch finanzielle und personelle Ressourcen, die zumeist knapp sind.

Beim maschinellen Lernen sind automatisierte Hyperparameter-Optimierungsverfahren weit verbreitet, um geeignete Trainingsparameter wie Lernraten und Batchgrößen zu finden. Sie reduzieren nicht nur den Arbeitsaufwand, sondern übertreffen meist auch die menschliche Konkurrenz in Geschwindigkeit und Qualität. Auf Basis ähnlicher Konzepte erreichten automatisch entworfene neuronale Netzarchitekturen im Jahr 2016 erstmals Spitzenleistungen bei modernen Aufgaben. Die Untersuchung solcher Prozesse, bekannt als neuronale Architektursuche (neural architecture search), gewann schnell an Interesse als mögliche Lösung für den Mangel an Arbeitskräften und als logischer nächster Schritt in der Entwicklung des maschinellen Lernens. Diese Arbeit konzentriert sich hauptsächlich auf zwei Aspekte der neuronalen Architektursuche:

Erstens analysieren und verbessern wir systematisch einen Baseline-Suchraum im Hinblick auf die Netzwerk-Latenz. Im überarbeiteten Suchraum entdecken wir Architekturen mit gleichwertiger Genauigkeit, die aber doppelt so schnell sind. In einem zweiten Schritt untersuchen wir, ob Suchraumdesigns auch automatisiert werden können. Der vorgeschlagene *Prune and Replace*-Algorithmus kann einen lose definierten Suchraum schrittweise durchsuchen und spezialisieren, auch wenn er wesentlich größer ist als zuvor. Aufgrund mehrerer technischer Optimierungen und Überlegungen benötigt die Suche dennoch weniger Zeit und ist in der Lage, bessere Architekturen zu entdecken.

Zweitens untersuchen wir Methoden zur Leistungsvorhersage in verschiedenen Szenarios. In einer groß angelegte Studie zur Hardwarevorhersage für verschiedene gängige Prädiktoren untersuchen wir im Detail, wie die multikriterielle Architektursuche durch verschiedene Faktoren wie die Qualität der Prädiktoren beeinflusst wird. Des weiteren evaluieren wir eine Änderung an sogenannten Supernetzwerken, einem weit verbreiteten Ansatz zur Vorhersage der Netzgenauigkeit. Die Änderung ist zwar derzeit schwer anzuwenden, führt aber zu einer durchgängig verbesserten Auswahl von Architekturen.

Abschließend stellen wir UniNAS vor, ein Framework zur Vereinheitlichung verschiedener Konzepte und Ansätze zur Architektursuche in einer einzigen Codebasis. Auf der Grundlage von Argumentbäumen (argument trees) können Experimente flexibel, sehr detailliert und sogar über eine grafische Benutzeroberfläche entworfen werden.

Acknowledgments

First and foremost, I want to thank Prof. Andreas Zell for his supervision of this dissertation, as well as the support, funding, and trust given to me over the years. I also want to thank Prof. Andreas Schilling for agreeing to be the second adviser, as well as Prof. Martin Butz and Prof. Jakob Macke for agreeing to be examiners.

I thank my former office-roommate Christian Geckeler, who was a great help in the FarmingIOS project, despite only working as a HiWi. I also want to thank my colleagues for the excellent time, be it supervising some lecture, eating together, or just having great conversations on private or scientific matters.

I am also grateful to Vita Serbakova, Klaus Beyreuther, and Uli Ulmer, who support everybody to keep the various bureaucratic and technical problems in check.

Last but not least, I am most grateful to my parents Irene and Peter, my sisters Marina and Nadine, and my better half Antje for their continuous moral and emotional support in my life.

Thank you.

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Outline and Contributions 3

2 Neural Architecture Search **5**
- 2.1 Timeline 5
- 2.2 Problem formulation 8
- 2.3 Categorizing Architecture Search 10
- 2.4 Efficient Architecture Search 11
  - 2.4.1 Optimal building blocks 12
  - 2.4.2 Super-Networks 13
- 2.5 Comparing Architecture Search 15
  - 2.5.1 Benchmarks 16
  - 2.5.2 Metrics 17

3 An efficient cell search space **19**
- 3.1 Introduction and Motivation 19
- 3.2 Related work 20
  - 3.2.1 Architecture Search 20
  - 3.2.2 Fast architectures 21
- 3.3 Method 23
  - 3.3.1 Identifying the NASNet design issues 23
  - 3.3.2 The ShuffleNASNet search space 24
- 3.4 Experimental evaluation 25
  - 3.4.1 Search 25
  - 3.4.2 Cell analysis 27
  - 3.4.3 Retraining 27
  - 3.4.4 Time performance 29
  - 3.4.5 Variations 29
- 3.5 Conclusions 31

4 Searching through vast architecture spaces **33**
- 4.1 Introduction 33
- 4.2 Related work 34
  - 4.2.1 Differentiable Architecture Search 34

4.2.2 Network pruning 36
4.2.3 Network morphisms 37
4.3 Method 39
4.3.1 Overview 39
4.3.2 Exploring vast operation spaces with morphisms 41
4.3.3 Operation similarity 43
4.3.4 Changing hardware requirements 45
4.4 Experiments and results 46
4.4.1 Search configuration 46
4.4.2 Retraining results 50
4.4.3 Resource analysis 50
4.4.4 Operation similarity analysis 52
4.4.5 Cell analysis 53
4.5 Conclusions 56

5 What to expect of hardware metric predictors in NAS 57
5.1 Introduction and motivation 57
5.2 Related work 59
5.3 Background 60
5.3.1 Multiple objectives 60
5.3.2 Differences between accuracy and hardware predictors 62
5.3.3 Model-based predictors 63
5.4 Methods 63
5.5 Predictor Experiments 64
5.6 Evaluating the predictor-guided architecture selection 68
5.6.1 Simulating predictors 69
5.6.2 Results 70
5.7 Discussion 72
5.8 Conclusions 74

6 Conditional super-network weights 75
6.1 Introduction 75
6.2 Foundations and Related work 76
6.3 Method 78
6.3.1 The problem 78
6.3.2 Conditional super-network weights 79
6.3.3 Search spaces 80
6.3.4 Evaluation metrics 82
6.4 Experimental evaluation 82
6.4.1 Search results 82
6.4.2 Resource analysis 85
6.4.3 Ablation study 86

6.5 How to find the improvement window . . . 87
6.6 Conclusions . . . 87

7 The UniNAS framework **89**
7.1 Introduction . . . 89
7.1.1 Available frameworks . . . 89
7.1.2 Common disadvantages of code bases . . . 90
7.2 Argument trees . . . 92
7.2.1 Modularity . . . 92
7.2.2 A global register . . . 93
7.2.3 Tree-based dependency structure . . . 94
7.2.4 Tree-based argument configurations . . . 94
7.2.5 Building the argument tree and code structure . . . 98
7.2.6 Creating and manipulating argument trees with a GUI . . . 98
7.2.7 Using external code . . . 100
7.3 Reproduced results . . . 101
7.4 Dynamic network designs . . . 102
7.4.1 Decoupling components . . . 102
7.4.2 Saving, loading, and finalizing networks . . . 105
7.5 Discussion and Conclusions . . . 106

8 Conclusions **107**
8.1 Summary . . . 107
8.2 Discussions and Future Work . . . 108

A What to expect of hardware metric predictors in NAS **111**
A.1 Encodings and Predictors . . . 111
A.2 Hyperparameters . . . 114
A.3 Selection of datasets . . . 115
A.4 Predictor fit time . . . 117
A.5 Approximating predictor mistakes . . . 117
A.6 Limits of MRA . . . 120
A.7 Simulation sanity check . . . 121

B Conditional super-network weights **123**
B.1 Super-network correlations . . . 123
B.2 Network designs . . . 124
B.3 Training and evaluation details . . . 125

C UniNAS **127**

Symbols **131**

Abbreviations **133**

Bibliography **135**

Chapter 1

Introduction

1.1 Motivation

In the not-so-distant past, the first successful form of artificial intelligence required an expert to specify a set of detailed decision-making rules. Equipped with if-then instructions and a body of knowledge, such expert systems emulated the explicit decision-making abilities of a human specialist. Carried by the enthusiasm of the initial successes of expert systems and computers in general, it was believed that letting a machine "describe what's on the image" could be done in a single summer project (Papert, 1966). It turned out that, while a human can easily find birds or other entities in an image, explicitly formulating the bird-detection rules is a complex problem.

Today, a fundamental component of machine learning is to learn such rules from data. By teaching a machine learning model where an image contains birds, training it to spot them by itself, we bypass our inability to describe the bird-detection rules. Due to the availability of computational resources, an abundance of training data, and the ever-increasing demand for new and improved applications, machine learning is nowadays found everywhere: Big corporations create extensive profiles of their users to recommend movies or advertisements that we may like. Photos are enhanced through filters, sometimes adding silly glasses, hats, or other accessories to one's person. Online services such as Google Translate or DeepL make automated translations to a variety of languages possible. Personal assistants understand instructions from spoken language, managing emails, and calendars for us. A smart home only heats when needed, reducing costs and the CO_2 footprint. Assistance systems in cars can automatically park correctly, switch lanes, or slow down if the road ahead is blocked. And much more.

Analogous to the unknown bird-detection rules, many details of such applications are not well understood either. Evidently, our complete understanding is not an essential part of their success. Thus, instead of designing and improving explicit rules, machine learning has primarily shifted to the design and improvement of systems that discover and optimize such rules automatically. In other words, to train a model.

While tremendously successful, this abstraction has its disadvantages. Aside from the computing power and data needed for the training, the thus added complexity is remarkable: The data often requires target values that the machine learning model has to

predict. It may be necessary to pre-process the data and account for distribution imbalances and rare events. Through various augmentation techniques, the amount of training data can be artificially increased. The model's design needs to promote a correct mapping from input data to target outputs, often under hardware- and task-related constraints. Additional techniques and model components regularize the training, ensuring that the model generalizes well to unseen data. In the case of the predominant gradient-based optimization, the model training is governed by sophisticated update rules and learning rate schedules. It may also be beneficial or necessary to choose specific hardware for the training process or distribute it across multiple devices. In short, the success of machine learning is governed by an intimidating wealth of techniques and configuration choices.

However, how can we find satisfactory or even optimal settings for a particular problem? Since an exhaustive evaluation is too expensive by many orders of magnitude, the typical answer is intuition and experience. Laboriously created and evaluated by domain experts, better models, training components, and configurations gradually replace their predecessors. Another answer is using hyper-parameter optimization methods, which use sophisticated search strategies and heuristics to optimize the training configuration automatically. Indeed, since configuring the discovery and optimization of bird-detection rules is a tremendously complex problem, a pragmatic approach is to guide their optimization with an automated meta-optimization. Such methods are part of AutoML, the growing field of Automated Machine Learning, the idea of *learning to learn*.

However, does adding another layer of complexity really improve the situation? Like the optimal bird-detection rules, it stands to reason that automatically designed training configurations are better than human-designed ones. And indeed, hyper-parameter optimization improves the experimental results in almost all cases. The ideal of AutoML goes even further: Every part of machine learning should be automatically optimized, even the meta-optimization itself. While AutoML is still in its early stages today, it has already influenced the wider field of machine learning significantly. Initial works that automatically design activation functions (Ramachandran *et al.*, 2018), gradient-descent update rules (Bello *et al.*, 2017) and especially data augmentations (Cubuk *et al.*, 2018), as well as their successors, have become part of many state-of-the-art records.

The topic of this dissertation is Neural Architecture Search (NAS), a large subfield of AutoML. With the focus on optimizing and designing neural network architectures, NAS takes care of the one thing that remains after the learning process is completed: the model. NAS aims to deliver high-quality networks for the ever-increasing variety of tasks and hardware platforms while reducing the user's efforts to a minimum. Given that neural networks are currently a dominant and popular form of machine learning models, NAS is applicable to almost any task.

Today, 28 of the top 100 ImageNet classification methods use EfficientNets[1], one of

[1] `https://paperswithcode.com/sota/image-classification-on-imagenet`, 08.10.2021

the most widely known NAS models. Corporations like Google[2] and Microsoft[3] already offer cloud-based AutoML solutions that require little previous knowledge of machine learning. In the future, AutoML may revolutionize and ultimately dominate the larger field of machine learning. The ability to solve many problems automatically, even if initially not as well as through an expert's work, will vastly accelerate the usage of machine learning in business applications and our everyday lives.

1.2 Outline and Contributions

After Chapter 1, the remainder of this dissertation is organized as follows. Chapter 2 starts by presenting the timeline for important NAS milestones and the presented works. NAS is then formulated as a meta-learning problem and broadly categorized along three axes: the search space, the search strategy, and the performance estimation strategy. The chapter concludes by presenting important efficiency tricks and methods for a fair evaluation used in the subsequent chapters and many NAS works.

Chapter 3 builds on ENAS (Pham *et al.*, 2018), the first economical NAS method. Due to the search space design, its discovered models are very complicated and slow. The chapter is based on the following publication where the search space is strategically redesigned, resulting in discovered models of equivalent performance and half the latency:

- K. A. Laube and A. Zell, ”ShuffleNASNets: Efficient CNN models through modified Efficient Neural Architecture Search”, 2019 International Joint Conference on Neural Networks (IJCNN), 2019, pp. 1-6, doi: 10.1109/IJCNN.2019.8852294.

Chapter 4 describes an approach to adjust the search space during the NAS process automatically. In contrast to methods where the search space is fixed, as in most modern NAS works, the proposed *Prune and Replace* method is capable of automated further specialization.

- K. A. Laube and A. Zell, ”Prune and Replace NAS”, 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA), 2019, pp. 915-921, doi: 10.1109/ICMLA.2019.00158.

Many modern NAS methods use prediction models to estimate architecture- and device-dependent metrics such as latency. We describe a large-scale study of different prediction methods in Chapter 5, and investigate how using such models affects the selection of architectures:

- K. A. Laube, M. Mutschler, and A. Zell, ”What to expect of hardware metric predictors in NAS”, accepted at AutoML-Conf 2022 `https://openreview.net/forum?id=HHrzAgpHUgq`

[2] `https://cloud.google.com/automl/`

[3] `https://www.microsoft.com/en-us/research/project/automl/`

In this work, Maximus Mutschler made valuable suggestions regarding the code implementation of the experiments.

Chapter 6 presents *conditional super-network weights*, a general approach to improve the accuracy predictions in different methods. While the results are promising, utilizing the concept in real-world applications is currently difficult.

- K. A. Laube and A. Zell, ”Exploring single-path Architecture Search ranking correlations”, initially submitted to ICLR 2021 (`https://openreview.net/forum?id=J40FkbdldTX`) but discontinued due to many similar papers emerging at the same time.
- K. A. Laube and A. Zell, ”Conditional super-network weights”, available on arXiv (`https://arxiv.org/abs/2104.11522`)

Due to the great variety of NAS-related methods, published code is highly diverse, fragmented, and generally hard to use. We present UniNAS in Chapter 7, a framework built with that issue in mind:

- K. A. Laube, ”The UniNAS framework: combining modules in arbitrarily complex configurations with argument trees”, available on arXiv (`https://arxiv.org/abs/2112.01796`) and GitHub (`https://github.com/cogsys-tuebingen/uninas`)

Chapter 8 then concludes this dissertation with a summary and outlook towards future work.

Finally, due to the thematic difference, a publication in which I assisted with the implementation is only mentioned here:

- M. Mutschler, K. A. Laube, and A. Zell, ”Using a one dimensional parabolic model of the full-batch loss to estimate learning rates during training”, available on arXiv (`https://arxiv.org/abs/2108.13880`) and presented as a poster at the NeurIPS 2021 workshop ”Optimization for Machine Learning” (`https://neurips.cc/Conferences/2021/Schedule?showEvent=21836`)

Chapter 2

Neural Architecture Search

Following the motivation for Neural Architecture Search (NAS) in Section 1.1, this chapter aims to provide a broad overview of the topic and to formally introduce NAS as a subfield of meta-learning. A short timeline of this recent field of study is presented in Section 2.1, followed by a formal description in Section 2.2. The major components of NAS methods are then introduced in Section 2.3, and two major efficiency tricks, which make the public usage of NAS possible, are detailed in Section 2.4. The chapter concludes with Section 2.5, which introduces the required benchmarks and metrics for a fair comparison between NAS methods.

2.1 Timeline

Architecture search only gained significant traction in the recent five years, initiating an ever-increasing stream of interest and publications in both academia and industry, as visualized in Figure 2.1. This section provides a summary of relevant milestones for this dissertation and puts them into context.

Although the idea of automatically designing neural networks dates back over three

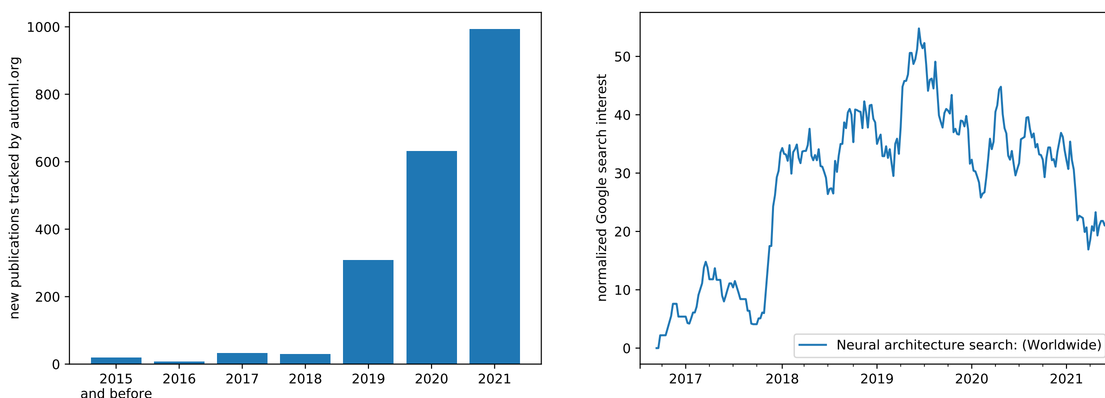


Figure 2.1: Interest in Neural Architecture Search over time. **Left**: number of publications per year, tracked by `automl.org`. **Right**: smoothed Google search trend.

decades (see Tenorio and Lee (1988), Kitano (1990) or Angeline *et al.* (1994)), the comparably tiny amounts of computational resources proved insufficient for wide-ranging studies and applications. The earliest dissertation-relevant work is Net2Net (Chen *et al.*, 2016). This network transformation method increases the sizes of convolution kernels in a trained network so that the network function remains identical (e.g., by padding with zeros). While its initial performance is thus unchanged, it can be improved further with additional training of the increased network capacity.

The pioneering work that led to NAS becoming a mainstream research topic is the automatic creation of network architectures for the CIFAR10 (Krizhevsky *et al.*, 2009) and Penn Treebank (Marcus *et al.*, 1993) benchmarks with competitive performance (Zoph and Le, 2016). Their architecture search method employs a recurrent network that outputs an encoded sequence translated into building instructions, from layer connections, kernel sizes, and strides in convolutions to activation functions. Each created child architecture is trained for 50 epochs, using its final validation accuracy to improve the recurrent network through reinforcement learning. Although the best architecture discovered performed better than the popular variants of ResNets (He *et al.*, 2016), training the recurrent network took the Google Brains team 800 GPUs running for 28 days, for a total of 22,400 GPU hours.

By reformulating the search problem to optimize a building block, rather than an entire network, a subsequent work cut the costs to around 2,000 GPU hours (Liu *et al.*, 2018). Since this widely used trick is utilized in Chapters 3 and 4, it is detailed in Section 2.4.1. The cost reduction also made further research directions possible, such as using evolutionary algorithms (Real *et al.*, 2018) instead of reinforcement learning, and a progressive architecture design (Liu *et al.*, 2018).

A further search space reformulation of even greater importance was introduced by ENAS (Pham *et al.*, 2018): All possible networks in the search space are considered sub-configurations of a single "over-complete" network, an idea that is elaborated in Section 2.4.2. When two disjoint networks are trained, both have to be trained from scratch, and their weights are irrelevant once evaluated. However, training a single network in an over-complete network also trains some weights for most other networks in the search space. They are no longer trained from scratch, and their training efforts are not lost once their architecture is evaluated. Consequentially, the massive reduction in training time for architecture evaluation enables this search method to run in a single day on a single GPU.

I excitedly started working with NAS and modern neural network design when I began my own Ph.D. time only three months later, resulting in my first publication based on ENAS and ShuffleNets (Ma *et al.*, 2018) in early 2019 (Laube and Zell, 2019b).

In parallel, the *over-complete* trick enabled another interesting development, aside from the computational efficiency: the usage of gradient-based optimization (DARTS, Liu *et al.* (2019)), which is significantly less complex than using hyper-parameter optimization via reinforcement learning or other external methods. A second fact that possibly contributed to DARTS' popularity was the decision to release public code based

on PyTorch (Paszke *et al.*, 2019), which was already in the process of replacing TensorFlow (Abadi *et al.*, 2016) in academia due to its simplicity.

My second publication directly manipulates the *over-complete* architecture. DARTS implicitly weighs the different paths in this architecture design, i.e., the candidate operations. By the process of *Prune and Replace*, Net2Net transformations are used to incrementally replace badly performing candidate operations and specialize the search space (Laube and Zell, 2019a).

However, the elegant design of gradient-based search methods comes with increased resource requirements. All architecture choices must be evaluated in every forward pass, making each update step several times more expensive. This initially led to the dependence on smaller proxy networks and proxy tasks, where architectures are discovered in a similar but less expensive environment. Cai *et al.* (2019) and Stamoulis *et al.* (2019) overcame this limitation, designing methods that discovered good networks directly even on ImageNet (Deng *et al.*, 2009).

However, gradient-based methods have another critical disadvantage: considering multiple optimization objectives such as accuracy, memory consumption, and latency. While optimizing for multiple objectives is generally possible by carefully weighing their components in the loss function (”scalarizing”), guaranteeing hard constraints was achieved only recently (Nayman *et al.*, 2021). In contrast, hyper-parameter optimization techniques like reinforcement learning and evolutionary algorithms face no such issues. They complete the architecture search process with an entire selection of potentially suitable candidates, providing a performance tradeoff among the different objectives. While Zoph *et al.* (2018) and Real *et al.* (2018) fully trained every considered architecture and spent thousands of GPU hours, Guo *et al.* (2020) use a single *over-complete* network as a cheap accuracy predictor. Once the model is trained, evaluating a specific architecture means simply selecting its corresponding subset in the *over-complete* network and computing a few batches on the validation dataset. Furthermore, their *Single Path One-Shot* method requires no more computational resources than training the most expensive architecture in the search space. It can be applied directly to the target task, data set, and network design. Performance tricks, such as the search for building blocks on proxy data, are not necessary.

Almost two years later, *Single Path One-Shot* is a widely used approach and the foundation for my recent works. One of its key aspects is the use of an *over-complete* network as a cheap prediction model. Even though many changes to the search space, evaluation method, and training schedule have been proposed, a fair comparison under the same circumstances was missing. We presented such a study in (Laube and Zell, 2021a), where we evaluate how different approaches affect its prediction quality. We then reused the developed techniques to present and evaluate *conditional super-network weights* in (Laube and Zell, 2021b). Unlike the weights normally used in *over-complete* networks (also named *super-networks*), our conditional weights can specialize towards the different candidate operations in previous layers. While this improves the selection of architectures consistently, the technique is currently difficult to apply and requires additional work.

In another study, we research which hardware metric predictors are the most useful for multi-objective architecture search (Laube *et al.*, 2022) and find that the widely used Lookup Tables ought to be replaced with MLPs. We also simulated how such predictors affect the selection of architectures and quantified how different evaluation settings affect the NAS results.

2.2 Problem formulation

Neural Architecture Search (NAS) is a subfield of Automated Machine Learning (AutoML), a subfield of Meta-learning. Meta-learning is intuitively described as "learning to learn", a process of finding the best model parameters (i.e. learning), but also to learn and improve the learning process itself. The meta-learning process is, therefore, a learning process on the metadata of machine learning experiments.

A particular instance of meta-learning is known to every machine learning practitioner: finding good hyper-parameters. In order to do so, the learning process has to be run multiple times, giving the person running these experiments performance feedback that may be used to guide the hyper-parameter choices in further attempts. A second widely known instance of meta-learning is transfer learning, where an already-trained machine learning model is finetuned on new data. While almost any kind of pre-training is helpful, this technique is most effective when both data sets are similar and the additional data plentiful.

To formally describe meta-learning, it is best to start with traditional machine learning:

Training a machine learning model Supervised learning encompasses a variety of machine learning tasks and data types, such as *image to class* in classification, *text to text* in translation, or *audio to text* in transcription. This is formally described as pairs of inputs x and targets y in a dataset $\mathcal{D} = \{(x_1, y_1), ..., (x_N, y_N)\}$. A machine learning model $\hat{y} = f(x; \theta)$ is then trained to predict the correct target $\hat{y} = y$ for a given input x by adapting the model parameters θ:

$$\theta^*(\Omega) = \arg\min_{\theta} \mathcal{L}(\mathcal{D}; \theta) \tag{2.1}$$

where $\mathcal{L}$ is the objective function that measures the loss of predicting $\hat{y}$ instead of y. The training process is further subject to many decisions Ω, such as the model design f, the parameter update rule, learning rate schedule, regularization techniques, and more.

Furthermore, in contrast to pure optimization, the goal of machine learning is to find θ^* with respect to a test dataset $\mathcal{D}^{test} \subset \mathcal{D}$, while the parameter training uses the disjoint training set $\mathcal{D}^{train} \subset \mathcal{D}$, $\mathcal{D}^{train} \cap \mathcal{D}^{test} = \varnothing$. While this increases the difficulty of the problem significantly, there is a practical reason for doing so: the trained model has to generalize to unseen data in order to make useful predictions $\hat{y}$, since $\mathcal{D}$ does not contain

all possible pairs x and y. If that were the case, a lookup-table over $\mathcal{D}$ would have already solved the problem.

Meta-Learning Conventional machine learning considers a pre-defined and fixed training configuration Ω, and is performed from scratch for every problem $\mathcal{D}$. In contrast, meta-learning learns the training parameters Ω, and is commonly formalized as a bi-level optimization problem. Such a problem consists of two hierarchically ordered problems, called *outer* and *inner*, which minimize $\mathcal{L}^{meta}$ and $\mathcal{L}^{task}$ respectively:

$$\Omega^* = \arg\min_{\Omega} \sum_{i=1}^{M} \left[\mathcal{L}^{meta}(\mathcal{D}^{valid\ (i)}; \theta^{*(i)}(\Omega)) \right] \tag{2.2}$$

$$\text{where } \theta^{*(i)}(\Omega) = \arg\min_{\theta} \mathcal{L}^{task}(\mathcal{D}^{train\ (i)}; \theta) \tag{2.3}$$

This general formulation considers Ω to be optimized on a set of tasks M, where each task $M^{(i)}$ is associated with a dataset $\mathcal{D}^{(i)}$. In this dissertation, $|M| = 1$ in all cases. Only a single training and validation dataset exist, which are disjoint subsets of the original training set $\mathcal{D}^{train}$.

This formulation emphasizes the hierarchical relationship of the inner and outer optimization problems: the inner optimization problem (Equation 2.3) is subject to a training configuration Ω, which is defined by the outer optimization problem (Equation 2.2). It is essential to realize that, in every update step of Ω, the currently optimal model parameters $\theta^*(\Omega)$ have to be computed anew. Due to the immense costs, practical solutions depend on approximations and efficiency tricks.

Neural Architecture Search Most AutoML and architecture search literature consider Ω to be mostly fixed, with only the model design being optimized. The optimal architecture a^* is then searched in the architecture space $\mathcal{A}$ and with otherwise fixed hyperparameters $\Omega_{nas} = \Omega \setminus \mathcal{A}$:

$$a^* = \arg\min_{a \in \mathcal{A}} \mathcal{L}^{nas}(\mathcal{D}^{valid}; \theta^*(a, \Omega_{nas})) \tag{2.4}$$

$$\text{where } \theta^*(a, \Omega_{nas}) = \arg\min_{\theta} \mathcal{L}^{task}(\mathcal{D}^{train}; \theta) \tag{2.5}$$

Since $\mathcal{A}$ is orders of magnitude smaller than Ω, AutoML and NAS can be considered simplified meta-learning. Intuitively, this simplification makes sense: if we were able to rank all architectures in the search space by their performance, small changes in Ω_{nas} (e.g., to the learning rate of the optimizer) are unlikely to impact this ranking significantly. It is computationally much more efficient to first find the architecture with

reasonable Ω_{nas}, and then optionally tune only Ω_{nas} in an additional round of hyper-parameter optimization for the final result.

2.3 Categorizing Architecture Search

Following Elsken *et al.* (2019), NAS can be broadly categorized by three dimensions. They are presented in Figure 2.2 and described below:

Search Space $\mathcal{A}$: NAS methods can only represent, train, evaluate, test, and discover the architectures in the search space $\mathcal{A}$. Since the search for optimal architectures in larger search spaces becomes increasingly difficult, the problem is often simplified by incorporating prior knowledge or intuitions about the expected solutions rather than using a space that contains everything imaginable. On the other hand, this human-introduced bias may also remove good and possibly novel candidates from the search space, preventing the NAS method from discovering them.

As the design of neural network architectures offers great possibilities, so does the design of search spaces. The currently predominant approach is to start with a specialized human-designed architecture and then extend the search space by questioning the details of its design. This may include the number of layers and their connectivity, such as skip connections and the number of channels. Other design options are the activation functions and operation details, such as kernel size and the number of groups in Convolutions.

As an example, the popular EfficientNet V1 models (Tan and Le, 2019) are, as many other NAS results, based on MobileNetV2 (Sandler *et al.*, 2018). The main changes of the EfficientNet-B0 architecture to MobileNetV2 are the kernel sizes and the expansion ratios of MobileNet's inverted bottleneck blocks, as well as the addition of Squeeze and Excitation modules (Hu *et al.*, 2018).

Search Strategy: The search space $\mathcal{A}$ is generally assumed to have exploitable characteristics that enable a methodical search to perform better than a random baseline. One prominent property is *locality*, which implies that any two architectures with a very similar design also perform similarly well (Ying *et al.*, 2019). Another is the existence of schemata, building blocks that are used in many top-performing candidate solutions.

There is a variety of optimization techniques capable of exploiting such properties. While the predominant approaches have historically been reinforcement learning (RL, Zoph and Le (2016); Zoph *et al.* (2018); Pham *et al.* (2018)), by now any common hyper-parameter optimization technique has been attempted. Some examples are progressive search (Liu *et al.*, 2018), evolutionary algorithms (EA, Real *et al.* (2018)), stochastic optimization (Xie *et al.*, 2018), bayesian optimization (White *et al.*, 2019), performance prediction (Wen *et al.*, 2020), and even a number of gradient-based methods (Liu *et al.*, 2019; Cai *et al.*, 2019; Stamoulis *et al.*, 2019; Hu *et al.*, 2020).

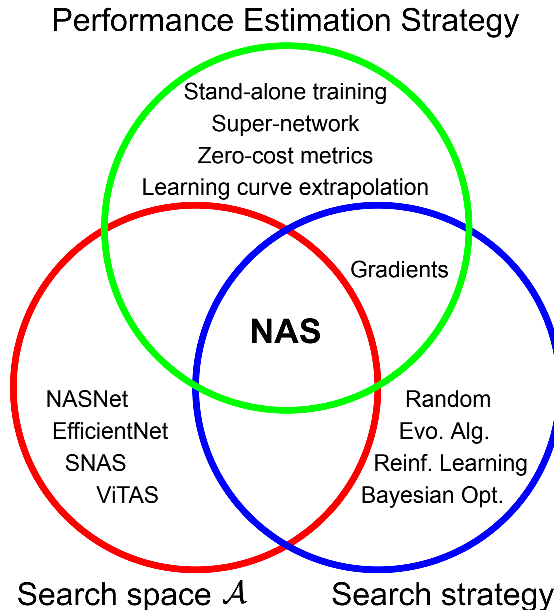


Figure 2.2: An overview of the NAS categories and some examples.

Performance Estimation Strategy: In order to traverse the search space $\mathcal{A}$ with a search strategy, it is necessary to evaluate the performance of each candidate architecture $a_{candidate} \in \mathcal{A}$. Although the approach of fully training $a_{candidate}$ from scratch is undoubtedly the most obvious and reliable, the computational demands of thousands of GPU days (Zoph and Le, 2016; Zoph *et al.*, 2018; Real *et al.*, 2018) render it impractical for everyday usage. Instead, the estimation correctness is relinquished for efficiency by using proxy metrics. These may include reduced network sizes, training epochs, or training set size, as well as learning curve extrapolation or performance prediction. A particularly efficient performance prediction technique is detailed in Section 2.4.2: the training of a single *over-complete* model which contains the weights of all architectures in the search space $a \in \mathcal{A}$ at once.

2.4 Efficient Architecture Search

Although the early modern NAS methods already achieved competitive performance, investing 20,000 GPU hours (Zoph and Le, 2016) into solving CIFAR10 is not broadly applicable. The following sections detail two changes that reduced this investment hurdle by orders of magnitude and enabled the widespread research and usage of NAS methods and results that we experience today.

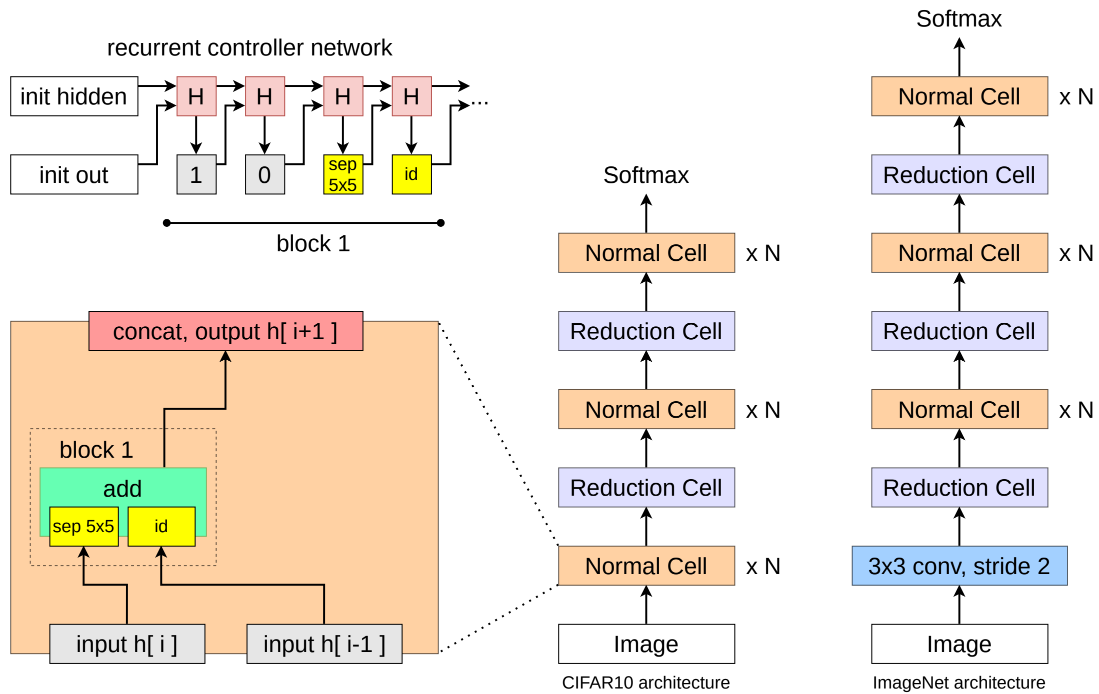


Figure 2.3: Designing cell-based neural networks. **Top left**: A recurrent network samples fixed-length codes that describe *blocks* in the *cell* below. Each *block* requires two input sources (gray) and one operation for each (yellow). **Bottom left**: The code is translated into a building block topology, called *cell*. Within are $B = 5$ *blocks* (only the first is visualized for clarity), which take inputs from either of the previous two *cell* outputs, or any previous *block* in the current *cell*. The *cell* output is then the concatenated output of all of its *blocks*. **Right**: *Cells* are stacked to create a network, depending on the data set.

2.4.1 Optimal building blocks

The first approach of letting a recurrent network design every detail of a network topology faced many difficulties, including an astronomical resource consumption (Zoph and Le, 2016). The output sequence of the recurrent network continued until a fixed stop signal was produced so that arbitrarily-sized child networks could be generated. Furthermore, the number of consecutive sequence values that define a computational graph node also varied since convolutions have more configurations (kernel size, stride, ...) than an identity function.

Zoph *et al.* (2018) simplified the problem in two ways, which are visualized in Figure 2.3: Firstly, instead of creating a complete network from scratch, networks are defined by the two building blocks *Normal Cell* and *Reduction Cell*. This concept was previously popularized, e.g., in the successful GoogLeNet (Szegedy *et al.*, 2015) and ResNet (He *et al.*, 2016) architectures, where stacking a few different block designs constitutes the network architecture. Secondly, only fixed-length sequences are used to

create a topology. Each cell in their search space consists of $B = 5$ blocks, and each block is defined by four numbers, resulting in a total of 40 numbers for both cell topologies. Unlike before, there exist only 15 different candidate operations such as Identity, 3×3 average pooling, 5×5 depthwise-separable convolution, or Zero. In combination, these two changes reduced the required architecture search time to 2,000 GPU hours while also achieving state-of-the-art results in image classification on CIFAR10 and ImageNet, and object detection on COCO (Lin *et al.*, 2014).

A crucial part of utilizing a cell-based search space, also called micro-level search space, is the ability to scale the network size. This enables reusing the results on other datasets and tasks and the actual architecture search to run on a much smaller network. While Zoph *et al.* (2018) train their search results on CIFAR10 by stacking $N = 6$ cells per stage for a total of 20 cells in the network, the actual search was performed using a much smaller proxy network of only eight cells ($N = 2$) and fewer channels. The changed search space thus enables the usage of a much cheaper performance estimation metric based on surrogate networks, datasets, and even tasks.

Although still widely used, cell-based search spaces lost some popularity since the first NAS methods proved that a search with fully-sized networks on target tasks is possible; in early 2019 (Cai *et al.*, 2019). It is currently agreed upon that early network layers benefit differently from convolution kernel sizes than later ones, especially when considering different hardware metrics (see e.g. Cai *et al.* (2019)). Instead of searching for a cell design that is used everywhere in a network, many modern spaces use proven human-designed building blocks from MobileNet V2 (Sandler *et al.*, 2018) or ShuffleNet V2 (Ma *et al.*, 2018). The NAS methods optimize the finer configuration details of each block in the network, such as expansion ratios, kernel sizes, activation functions, and attention mechanisms.

2.4.2 Super-Networks

The primary intent of super-networks is to share (nearly) all trainable weights among different network architectures. The concept has several names in literature, such as super-network, one-shot model, or over-complete graph, which are all used synonymously in this dissertation.

Theory and Implementation A common way to describe neural networks is based on graph theory, considering the network a directed acyclic graph (DAG) of computations. The vertices and edges describe the operations and information flow, respectively. This view was used very explicitly in TensorFlow V1 (Abadi *et al.*, 2016) and is still at the core of all modern deep learning frameworks, but often hidden behind more convenient high-level perspectives. Pham *et al.* (2018) extend this view by considering each specific architecture as a graph subset of the single, over-complete graph that contains all possible architectures: the over-complete super-network.

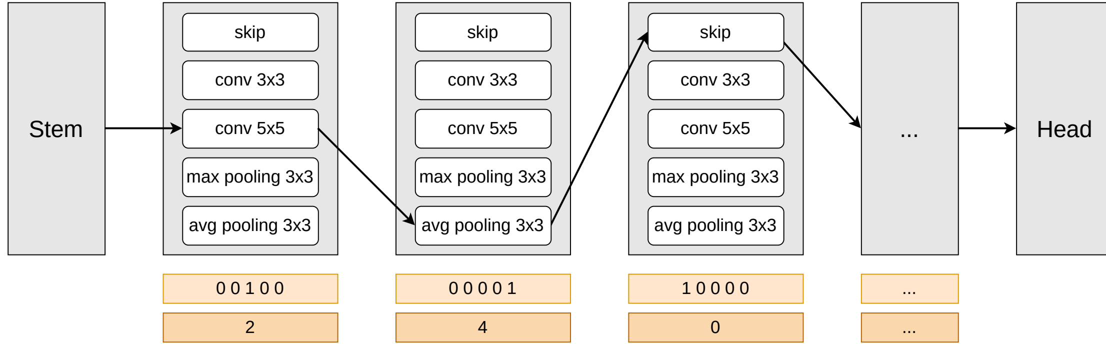


Figure 2.4: A sequential super-network with three displayed layers and five candidate operations in each. The connecting arrows constitute one specific architecture in the search space, a subset of the over-complete computational graph. Other possible representations of a specific architecture are layer-wise one-hot gates or the indices of the currently selected operations, here $[2,4,0,\ldots]$. The network weights are in the graph vertices and thus shared among all possible architectures.

An example is visualized in Figure 2.4: The white boxes are the candidate operations, grouped by which layer they correspond to. In this design, a particular architecture (marked by the edges connecting operation-vertices) consists of precisely one operation per layer, chosen among five candidates. Since the network weights are placed in the vertices, the displayed subgraph (network) would share the weights in the first layer (a Conv 5×5 operation) with every other subgraph that also uses this particular candidate operation. For argument's sake, consider an over-complete network with ten layers and any specific architecture a_{cur} in it. This super-network contains a total of 5^{10} valid unique architectures, of which four are identical to a_{cur} except for the first operation. Indeed, $4 \cdot 10 = 40$ unique networks differ from a_{cur} in only a single operation, which is also called an edit-distance of 1 (Ying *et al.*, 2019).

While the example in Figure 2.4 is straightforward, more complicated super-network designs are also possible. Covering the previously described cell-based search (see Figure 2.3) requires operations to consider multiple input sources that may produce tensors of different shapes. There are two easy solutions: Firstly, each candidate operation has one set of weights for every possible input source with a different tensor shape. By using the correct weights per input tensor, an output with consistent size can be guaranteed. Secondly, depending on the candidate operation, it may be possible to combine different sets of weights into one. Suppose two image-tensors have the same spatial size but 128 and 256 channels respectively, and the candidate $K_1 \times K_2$ convolution has weights of shape $[c_{out}, c_{in}, K_1, K_2]$, with $c_{in} \in \{128, 256\}$. By considering the smaller weight tensor a subset of the larger one, i.e. $[c_{out}, 128, K_1, K_2] \subset [c_{out}, 256, K_1, K_2]$, weight sharing across input sources becomes possible as well. Finally, operations like Skip or Pooling can only change the input tensor size in limited ways. Adding a linear convolution operation (no activation function) to them may be necessary to guarantee the shape of their output.

Disadvantages Despite ongoing efforts (see Sciuto *et al.* (2019) and Yu *et al.* (2020b) for examples), the impact of using super-networks in architecture search is still poorly understood. Early NAS methods have been shown to suffer from a *performance-gap* between networks trained as a subset of a super-network and those trained independently, which following works mitigate with different approaches (e.g., Chen *et al.* (2019)). Furthermore, super-networks require well-defined search spaces in which all possible connections and candidate operations are known from the start. Finally, early gradient-based approaches required executing every possible operation in the graph for every network forward pass, which necessitated the use of small proxy networks (Liu *et al.* (2019), also see Section 2.4.1). This issue has been solved by several different approaches (see e.g. Chen *et al.* (2019), Dong and Yang (2019), or Hu *et al.* (2020)) at the cost of increased complexity.

Advantages The supreme advantage of super-networks is the incredible cost-efficiency. Previous NAS methods required the evaluation accuracy of several thousand trained candidate architectures to guide the search, which needed to be trained independently on hundreds of GPUs over days. The super-network approach replaces this extremely costly performance estimation strategy with a much cheaper one: training only one super-network once and evaluating any candidate in the search space by simply running the appropriate subgraph on the validation data. Pham *et al.* (2018) thus reduced the previous search costs of 2,000 GPU hours to 12, albeit also using only eight of the 15 candidate operations. While the super-networks' specific training and evaluation details are a highly contested research question, their concept has become a central component in most modern NAS methods.

A significant factor for the weight sharing efficiency is that, while the search space grows exponentially with the number of candidates, the super-network increases only linearly in size. Even though the super-network in Figure 2.4 contains 5^{10} possible architectures, only $5 \cdot 10$ nodes with weights are required and need to be trained. Since the major memory consumption during network training stems from storing all intermediate output tensors of the graph nodes, not the network's weights, the increased size of a super-network alone generally poses no problems.

2.5 Comparing Architecture Search

As a new discipline that received much attention quickly, many initial NAS works lack the baselines and practices for fair comparisons found in more mature fields of study. Common issues are missing (random) baselines, lack of ablation studies, unclear hyper-parameter choices, the unavailability of the code for reproduction, and even unfair comparisons. Attempts to compare published methods fairly are "frustratingly hard" (Yang *et al.*, 2019), and show that many early methods perform no better than a random baseline (Li and Talwalkar, 2020)).

However, progress is being made. Lindauer and Hutter (2020) propose a set of best practices for NAS research, and an increasing number of modern NAS publications provide nearly-standardized performance metrics on newly available benchmarks.

2.5.1 Benchmarks

Benchmarks and baselines are indispensable tools for fair comparisons. NAS benchmarks contain important statistics for many architectures in a specific search space, such as the test loss or the number of parameters. Ying *et al.* (2019) took a critical first step in this direction, compiling the results of over 5 million trained models in a public dataset named NAS-Bench 101. Their search space contains roughly 423k unique architectures, which have all been trained on CIFAR10 multiple times.

By querying the benchmark for result metrics of a specific architecture, the otherwise costly performance estimation metric can be performed for almost free and without any random training factors. Hyper-parameter optimization frameworks can thus perform a reproducible architecture search in mere seconds. Methods that require a super-network (e.g., gradient-based) still need to train one, but can at least look up the performance of the discovered architecture.

Additionally, knowing all results of an entire search space provides further insights. One of many significant findings is that the correlation between validation and test accuracy is extremely high ($r = 0.999$), showing that overfitting on validation metrics is very unlikely. They also showed that very similar network structures, measured by edit-distance, also have a very similar performance. This effect of *locality* vanishes with increasing distance and is no longer noticeable after around six changes. In practice, that shows that many networks of high quality generally surround top-performing ones.

Other benchmark datasets followed, with different search spaces and foci. Dong and Yang (2020) published NAS-Bench 201, which provides detailed statistics throughout the training process for each of the 15625 networks in the search space on three vision datasets. Li *et al.* (2021a) extended NAS-Bench 201 with detailed hardware metrics on different devices, such as Pixel smartphones or Raspberry Pis, in their HW-NAS benchmark. The recent TransNAS-Bench 101 (Duan *et al.*, 2021) provides performance statistics of 7.3k architectures across seven tasks such as image classification, object detection, or pixel-level prediction. While all benchmarks mentioned above provide tabular lookup data, Siems *et al.* (2020) experiment with surrogate models for predicting metrics correctly in their NAS-Bench 301, a promising avenue for much larger search spaces.

A curious development following NAS-Benchmarks are NAS methods that require no training at all (e.g., Mellor *et al.* (2020)). The authors find that probing uninitialized networks can indicate the final test performance and even compete with established NAS approaches. White *et al.* (2020) find that even local search can be competitive on NAS benchmarks. However, neither training-free nor local search approaches are competitive with other NAS methods in larger search spaces, indicating the need for improved benchmarks.

2.5.2 Metrics

The intention behind NAS methods is to find the best-performing architecture. A stable and correct ranking of architectures is thus more important than predicting each architecture's performance correctly. The commonly favored ranking correlation metrics are *Spearman's rank correlation coefficient* (SCC, ρ) and *Kendall's Tau* (KT, τ). Both correlations share the property that they are bounded in the interval $[-1, 1]$, and that a value of 0 indicates statistical independence between the measured variables X and Y.

In the case of architecture search, X and Y are the performance predictions and ground-truth accuracy values, respectively. The true function $f(a)$, which maps every architecture $a \in \mathcal{A}$ to an accuracy value, is approximated by a prediction function $f_p(a)$ such as a super-network. Using the unseen architectures $\mathcal{A}_{test} \subset \mathcal{A}$:

$$X = \{f_p(a) | a \in \mathcal{A}_{test}\}, \quad Y = \{f(a) | a \in \mathcal{A}_{test}\} \tag{2.6}$$

The Spearman's correlation is defined as the Pearson correlation coefficient between rank variables. The variables X and Y are first converted to ranks R_x and R_y, on which the Pearson correlation is computed:

$$\rho_{X,Y} = \frac{cov(R_x, R_y)}{\sigma_{R_x} \sigma_{R_y}} \tag{2.7}$$

This metric measures how X and Y are monotonically related, with values close to -1 or 1 indicating a strong relationship.

In contrast, Kendall's Tau counts how often all pairs of observations (x_i, y_i) and (x_j, y_j)

1. are concordant, agreeing on a sorting order
   ($x_i < x_j$ and $y_i < y_j$ or $x_i > x_j$ and $y_i > y_j$)
2. are discordant, disagreeing on a sorting order
   ($x_i < x_j$ and $y_i > y_j$ or $x_i > x_j$ and $y_i < y_j$)
3. are neither

Kendall's Tau is then calculated by their difference and normalized by the number of possible different pairs:

$$\tau = \frac{(num\ concordant) - (num\ discordant)}{\binom{n}{2}} \tag{2.8}$$

τ ranges from -1 in perfect disagreement to +1 in perfect agreement, and is close to zero when X and Y are independent. Both ranking metrics are visualized in Figure 2.5.

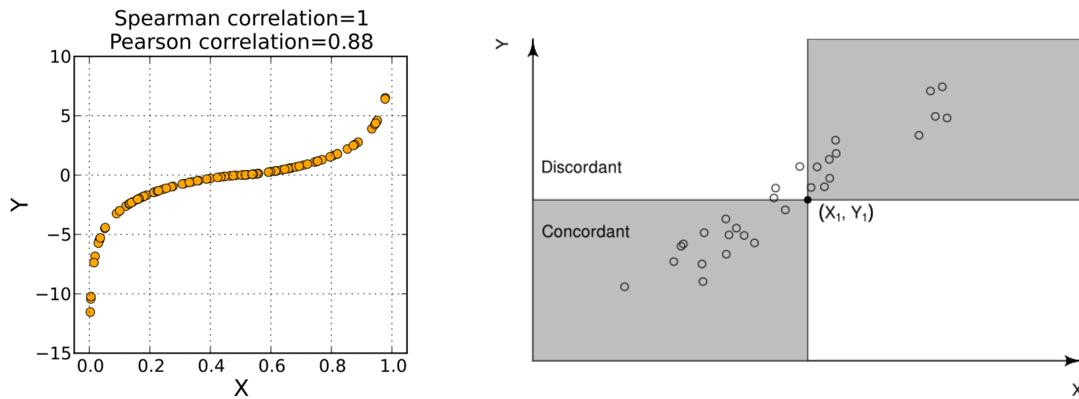


Figure 2.5: **Left**: The Pearson and Spearman correlation metrics on a sequence of monotonic points. **Right**: Kendall's Tau visualized. Image sources: Wikipedia (2021).

However, estimating the ranking correlation requires the availability of a NAS benchmark and the ability to evaluate any architecture in the search space. The first requirement was not fulfilled for many early NAS works since simply no benchmark existed. The second requirement may not be possible for the NAS method in question, especially if it uses gradients for a continuous transition into a finalized result state. Nonetheless, the ability to quickly query how well the discovered architecture ranks in the search space is a huge benefit of benchmarks. If such benchmark data is not available, it is necessary to perform the search multiple times, report the average results, perform ablation studies, and compare with random baselines and other methods in mostly identical environments.

Chapter 3

An efficient cell search space

In this Chapter, we redesign the NASNet cell search space following the design guidelines of ShuffleNet V2 (Ma *et al.*, 2018). The discovered architectures in this search space are less complex and two times faster while maintaining their baseline's parameter efficiency and accuracy. Most aspects of this chapter were described and published more compactly in Laube and Zell (2019b).

3.1 Introduction and Motivation

For the first time, automatically designed neural network architectures convincingly outperformed their hand-crafted competition in image classification and object detection (Zoph and Le, 2016; Zoph *et al.*, 2018). Although NAS was prohibitively expensive, *Efficient Neural Architecture Search* (ENAS, by Pham *et al.* (2018)) reduces the costs to mere hours on a single GPU, thus facilitating its extensive use in research and industry. Admittedly, the thereby discovered architectures have a severe flaw: their inference speed.

A fast execution time, however, is critical in many modern applications. A self-driving car at speed has to detect pedestrians in time, and failures in control systems have to be recovered before any damage is caused. The network designs for such tasks follow very different design principles: they have to be fast, often at the cost of correctness and operate with limited available computing power or memory. Another example of extensive research is not quite as dangerous but subject to the same conditions: image classification on smartphones. The two predominant lines of this research are Mobilenet V1 and V2 (Howard *et al.*, 2017; Sandler *et al.*, 2018) as well as ShuffleNet V1 and V2 (Zhang *et al.*, 2018; Ma *et al.*, 2018), both of which also became increasingly important in NAS designs shortly after.

To apply NAS in low-latency environments, a complete redesign of the architecture search space is unavoidable. Using a fully sized network during the search phase is not yet possible, so the NAS method must search for optimal building blocks instead, using smaller and cheaper proxy networks. We follow the ShuffleNet V2 guidelines for fast and efficient architectures, creating a new NAS search space in which we find our new and improved models: ShuffleNASNets.

3.2 Related work

There are two main categories for related work: inexpensive NAS approaches optimizing the network accuracy and the manually designed patterns that improve execution speed.

3.2.1 Architecture Search

A fundamental requirement for widely applicable NAS is a fast and economical architecture search method. Elsken *et al.* (2018) add new connections and operations to a network in an iterative fashion based on greedy hill-climbing and random mutations. Brock *et al.* (2018) use a hyper-network to predict the weights of candidate architectures in the search space, thus significantly reducing the training overhead. However, neither method reaches the performance and cost-effectiveness of *Efficient Neural Architecture Search* (ENAS) by Pham *et al.* (2018), the first pillar of this work. In contrast to its expensive predecessors (Zoph *et al.*, 2018; Real *et al.*, 2018) that required thousands of GPU hours to discover the architectures that dominated image classification and object detection, ENAS takes half a GPU day. Even though ENAS performs slightly worse and uses a smaller search space, the economic effectiveness makes it an attractive foundation.

The primary invention of ENAS is the cheap architecture performance estimation based on a weight-sharing super-network, a concept that has been detailed in Section 2.4.2. Fundamentally, all candidate architectures trained by ENAS are subsets of a single over-complete computational acyclic graph. Akin to a predictor, this graph (also called super-network) is then used as an inexpensive performance estimator for any graph subset (architecture). To select an architecture for training or validation, ENAS generates a sequence of integers using a recurrent controller network of 100 LSTM units (Hochreiter and Schmidhuber, 1997). Interpreted as graph-edges, this sequence uniquely encodes the topology of one specific architecture. Its validation performance is used as a guiding signal to train the controller network using reinforcement learning (Williams, 1992) so that prospective sampling is more likely to discover well-performing architectures. Specifically, ENAS trains one particular architecture for an epoch and evaluates another ten at its end. Their estimates are used to train the controller; and the best candidate is selected for training in the next epoch. At the end of the search process, the candidate with the highest estimated validation accuracy is considered the search result.

While an overview of cell-based search spaces and their advantages has already been given in Section 2.4.1, we briefly review the essential details again: The NASNet search space contains various possible designs for building blocks, called *cells*, which are stacked to create task-specific network architectures. Figure 2.3 visualizes the ENAS search result and the cell embedding in an image classification network. Each cell contains $B = 5$ blocks that take two inputs from two previous cells or previous blocks in the same cell. One operation is applied to each input, their sum forming the block output. All block outputs that other blocks have not used are concatenated and used as cell output, with an optional 1×1 convolution that can adjust the number of feature channels. As visualized,

ENAS stacks a total of six *Normal Cells* and two *Reduction Cells* in the search network, which are used to respectively keep and reduce the spatial size of the image tensors.

3.2.2 Fast architectures

While the aforementioned NAS-based architectures are often remarkably efficient concerning accuracy or parameters, their inference speed is terrible. Although attempts to search for fast and even device-aware architectures existed (e.g., Dong *et al.* (2018)), they were not economical at that time.

Instead, the predominant architectures in this field were hand-crafted with specific care for low memory footprints and fast execution times. MobileNet V2 (Sandler *et al.*, 2018) features the famous *MBConv* block, an inverse bottleneck structure that temporarily expands the number of channels using a 1×1 convolution by up to six times. ShuffleNets (Zhang *et al.*, 2018; Ma *et al.*, 2018) introduced a shuffle unit that inexpensively rearranges the order of channels in a tensor. This concept is only meaningful when combined with operations that are applied to subsets of channels, as visualized in Figure 3.1: Given C feature channels, a 1×1 convolution spanning them all has C^2 parameters and C^2 multiplications per image pixel. When separated into G groups, the costs reduce to $G \cdot (C/G)^{C/G}$, which is considerably less expensive. The price of such grouped convolutions is a reduced information flow, which makes their consecutive application disadvantageous. However, by rearranging the channels, shuffle units improve the information flow. In addition, not applying any non-linear functions to some channels implicitly creates shortcut connections of different lengths across the layers. Furthermore, Ma *et al.* (2018) revealed bottlenecks for inference speed in a series of experiments, which we made the second pillar of our work:

- **G1**: Equal channel width minimizes memory access costs.
- **G2**: Excessive group convolutions increase the memory access costs.
- **G3**: Network fragmentation reduces the degree of parallelism.
- **G4**: Element-wise operations are non-negligible.

Most architectures violate one or more of these rules, impairing their inference speed, especially with constrained memory or computing power. For example, inception modules (Szegedy *et al.*, 2015) concatenate several paths, violating G3. Ma *et al.* find that the inference speed on GPUs can drop by a factor of 1.6 when using two parallel paths and by 3.5 when using four, though the drop is much less severe on ARM hardware. ResNet (He *et al.*, 2016) bottleneck blocks and the inverse bottleneck blocks of MobileNet V2 violate G1. Both change the channel number by a factor of around four and then restore it to the original count with another convolution. Almost all architectures violate G4 by adding many biases, applying many activation functions, and using skip connections that contribute to G3 and G4. In sum, MobileNet V2 spends almost half as much GPU inference time on element-wise operations as on convolutions.

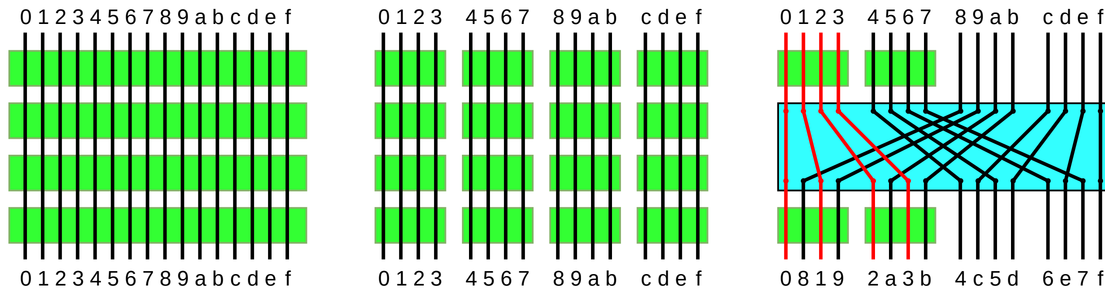


Figure 3.1: Three variants of convolutions (green) applied to the 16 channels numbered 0 to f. **Left**: Since each convolution considers all channels, the operations are expensive and require many parameters but have a good information flow. **Center**: Group convolutions consider only small groups of channels at a time. They have a limited information flow but are much cheaper. **Right**: Shuffle units cheaply rearrange the channels, enabling the use of successive cost-effective group convolutions with an improved information flow. Additionally, half of the channels skip these operations, providing shortcut paths of different lengths.

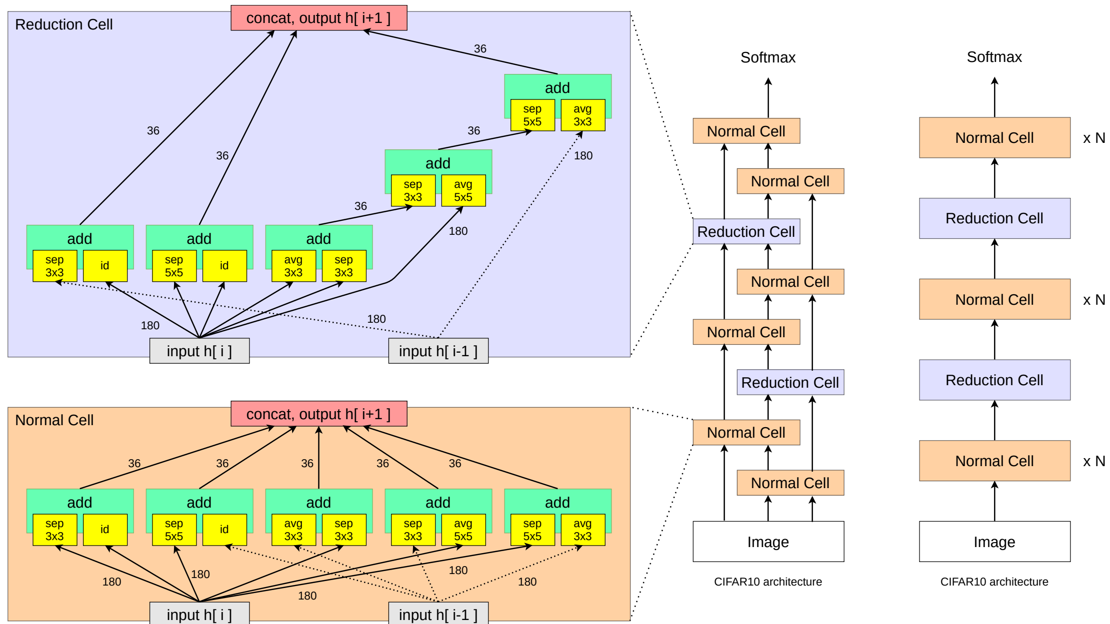


Figure 3.2: **Left**: The Normal and Reduction Cells discovered by ENAS. **Right**: Two representations of the network embedding these cells. While the right representation is the commonly used one, it is also simplified. The left one describes the connectivity better, since each cell takes the outputs of two prior cells. The search network uses $N = 2$, the fully-sized evaluation model $N = 5$. The numbers in the cells are example channel sizes, highlighting some issues with the design.

3.3 Method

3.3.1 Identifying the NASNet design issues

The first step of improving a design is to identify its key issues. Since the NASNet search space was not designed with inference time in mind but correlates speed with a low number of parameters, their resulting architectures severely violate the guidelines G1 to G4. We list **s**tructural issues that lead to **s**low inference time based on the ENAS architecture in Figure 3.2 as points **S**:

- **S1**: NASNet cells are based on inception modules and are highly fragmented (G3).
- **S2**: Some computational paths, such as convolutions of the same size, could be combined (G1, G3). The 3×3 avg pooling in the Normal Cell is even redundant.
- **S3**: A total of 10 operations are applied in $B = 5$ blocks, which each need to sum their operation results (G4). To maintain the cell channel count after concatenating the block outputs as cell output, every operation has to reduce the channel count by a factor B (G1), unless the input already has that size. Since some operations (e.g., pooling or identity) preserve the channel count, they require an additional linear 1×1 convolution (G1, G4).
- **S4**: A detail that is hidden for visual clarity is that every single displayed separable convolution operation actually consists of two stacked ones (G2, G4).
- **S5**: The network fragmentation is not only limited to the cell designs. Since each cell receives the outputs of two prior cells, parallel paths are necessary (G3).

Furthermore, there are also many drawbacks and points of **c**omplicated design **C**. Some are especially apparent when constructing the super-network:

- **C1**: Cells receive the outputs of two prior cells, which may have different spatial and channel sizes. They need to pre-process each input tensor with an additional convolution to match them (not visualized).
- **C2**: All operations may receive their input tensors from previous cells or blocks, which have different channel sizes. ENAS makes a copy of each operation for each possible channel count and activates only the needed one.
- **C3**: The cell output concatenates only the unused block outputs, which varies depending on the currently active graph subset (i.e. architecture). While the pre-processing convolutions (C1) in both following cells could compensate for that, this solution makes learning more difficult. ENAS adds an additional linear 1×1 convolution instead (not visualized).
- **C4**: Since it is necessary to reduce the channel size after an operation, a genuine skip connection is not possible.
- **C5**: The only way to reduce the cell complexity is the Zero operation, which can be used with a second operation in a block. However, not using an entire block (both operations are Zero) this way is hardly beneficial since the block results are concatenated. The influence of all other blocks on the cell output stays the same.

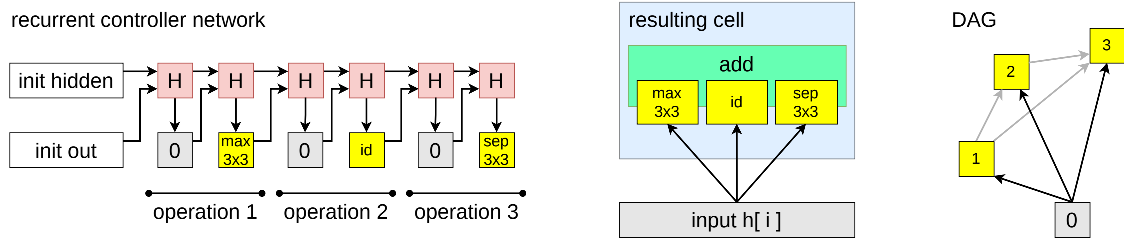


Figure 3.3: An example sampling process for $B = 3$ blocks. **Left**: The controller network generates B pairs of indices per cell, which are interpreted as input sources (gray) and applied operations (yellow). **Center**: The sequence of indices is translated into a cell topology. All operations take input 0, which is the cell input. **Right**: The current cell graph is only a subset of the super-network. Following the controller's instructions, the connections to the cell inputs are currently active (black), the others to blocks within the cell are not (gray).

3.3.2 The ShuffleNASNet search space

As the analysis in Section 3.3.1 shows, the search space has to be redesigned from scratch. The first major change is that each cell has only a single input, which reduces the degree of parallelism in the network (G3, S5, C1). We instead use shuffle units to provide skip connections of different lengths, regardless of cell design. While NASNet uses B blocks with two operations each, we use only $B = 5$ operations in total and make their sum the cell output, as visualized in Figure 3.3. This significantly reduces the total number of operations and fragmentation (G3, S3) and solves the issue of having a variable amount of tensors in concatenations (C3). Since the new architecture is consequently encoded in shorter sequences, the sampling controller network should also learn the problem better.

Each ShuffleNASNet cell operates only on half of the current image tensor channels and does not change that amount anywhere. This enables using identity functions as genuine skip connections (C2, C4) and reduces complexity when used before or after any other operation (C2, C5). Since doing so effectively reduces the number of operations B_n, the corresponding search space $\mathcal{A}_n$ fully contains all smaller spaces and can be regarded as an upper bound of cell complexity:

$$B_1 > B_2 \implies \mathcal{A}_1 \supset \mathcal{A}_2 \tag{3.1}$$

We furthermore add a $1{\times}1$ convolution to the pool of ENAS' candidate operations, which enables reorganizing the channels before they are shuffled. The six available candidate operations are:

- $1{\times}1$ convolution
- Separable $3{\times}3$ conv., dilation = 1
- Separable $5{\times}5$ conv., dilation = 1
- Max. $3{\times}3$ Pooling
- Avg. $3{\times}3$ Pooling
- Identity or Factorized Reduction

Identity is the only candidate operation that can not easily reduce the spatial size of image tensors, as required in reduction cells. Following ENAS, we use Factorized Reduction operations (1x1*) in such cases: two linear 1×1 convolutions with stride two are applied spatially shifted, their results are concatenated. This creates a grid pattern, which is visualized in Figure 3.4.

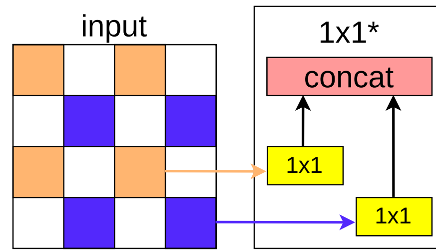


Figure 3.4: Factorized reduction

It is unclear whether adding BatchNorm (Ioffe and Szegedy, 2015) is beneficial to the networks. While it is known to improve the performance in deep architectures, it also adds additional element-wise operations (G4) that may not be necessary.

While the resulting search space will still produce fragmented cells, the overall fragmentation, amount of operations, and complexity have been significantly reduced. Consequentially, the search space is also much smaller but still far too huge to evaluate exhaustively.

3.4 Experimental evaluation

3.4.1 Search

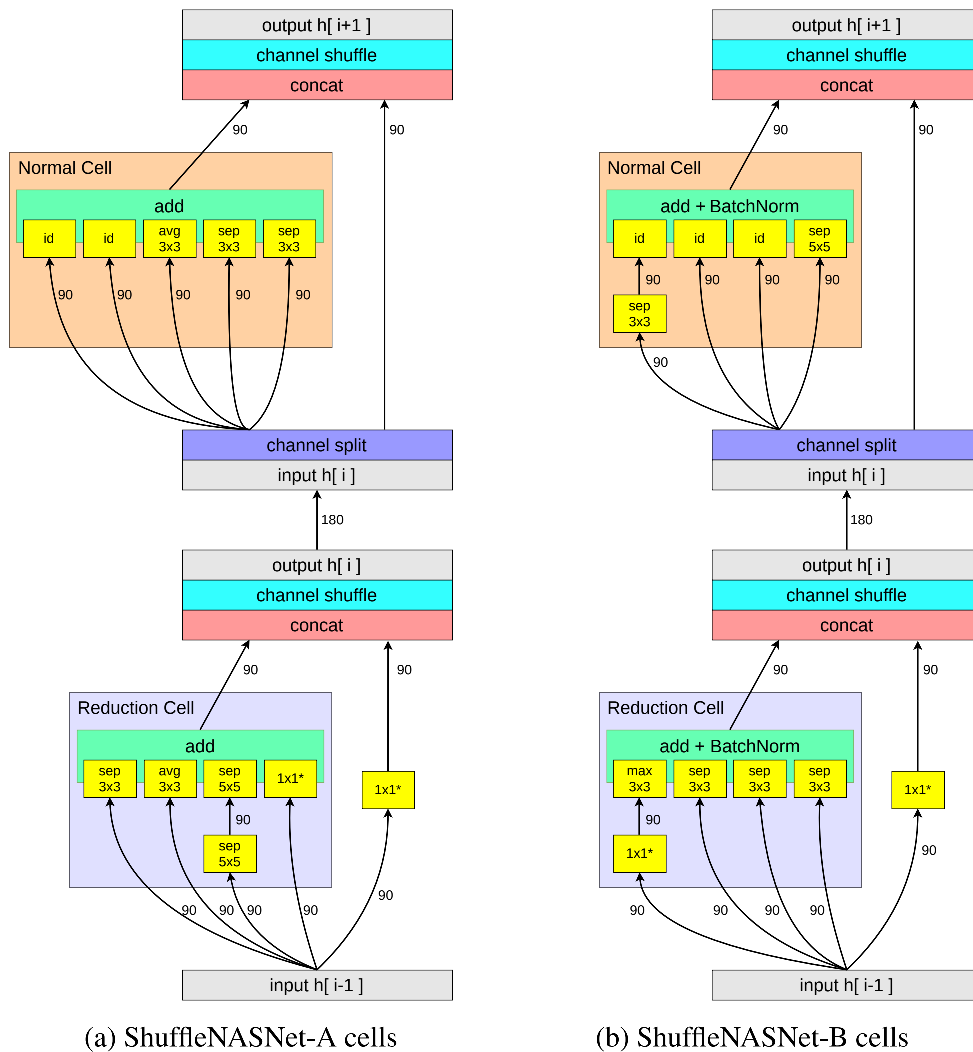
Except for the changed search space, our experiments closely follow the ENAS baseline. We search for the optimal topologies of Normal and Reduction cells in a super-network that contains all possible configurations as graph subsets. This search network stacks a total of six Normal and two Reduction cells with shared topology but differing weights and tensor sizes. The network weights θ are trained on CIFAR10 Krizhevsky *et al.* (2009), which consists of 60,000 32×32 pixel color images, each belonging to one of 10 classes. Ten thousand images belong to the test set, and we reserve another 5,000 as a validation set to train the controller network's parameters ω.

The ENAS search process is a significant simplification of the original meta-learning stated in Equations 2.4 and 2.5 since it interleaves the training of network and controller parameters (θ and ω respectively, the current architecture a depends on ω). Following ENAS, we use a controller to sample ten candidate architectures per epoch and evaluate them on the validation set. Their performance feedback is used to train ω using reinforcement learning (Williams, 1992) and the ADAM optimizer (Kingma and Ba, 2015), the best architecture a_{best} is used to train θ with SGD for one epoch. Both optimizers are subject to cosine annealing with warm restarts (Loshchilov and Hutter, 2017). All architectures share θ via the super-network, so the training of any specific a_{best} also benefits most other architectures.

One entire search process requires only eight to ten hours on a single Nvidia 1080 Ti

GPU, with some randomness that depends on the selected architectures in each epoch. Since it is unclear whether BatchNorm is beneficial, we run separate search processes for cells both with and without.

The controller network generates several candidate architectures during the search process. We consider the best architectures in the final epoch of training and sampling and the architecture with the best validation accuracy during the entire search process. ShuffleNASNet-A is the best performing design without BatchNorm, selected from the all-time highest validation accuracy during the search process. It is slightly smaller than the BatchNorm-using ShuffleNASNet-B, which was selected from the final controller suggestions in a separate search process to ShuffleNASNet-A.



(a) ShuffleNASNet-A cells (b) ShuffleNASNet-B cells

Figure 3.5: The two discovered ShuffleNASNet variants embedded in network structure.

3.4.2 Cell analysis

Both resulting ShuffleNASNet variants are visualized in Figure 3.5, embedded in ShuffleNet's structure of channel splitting, concatenating, and shuffling. As designed, the cells are much less complex and fragmented than NASNet cells and do not change the number of channels anywhere.

We find that the Normal Cells of both discovered architectures each use two identity paths. This is interesting, since half of the channels already bypass the entire cell in ShuffleNet-like architectures. Also adding two skip connections in the cell may appear redundant at first glance. However, the channels will be concatenated with the skip path and shuffled, and the sum operation within is not weighted. The discovered topology thus combines a ShuffleNet and ResNet approach, and puts additional emphasis on the skip information not being outweighed four-to-one.

Another interesting property of ShuffleNASNet-B is an identity function after a convolution in its Normal Cell. This arrangement effectively reduces the cell complexity to $B = 4$ blocks and should also improve the inference time, despite not being specifically optimized for. In a re-built inference network, this operation can be removed.

3.4.3 Retraining

The discovered cells designs can now be used in task-specific networks. We train models on CIFAR10 and CIFAR100, which have 10 and 100 classes, respectively, but are otherwise very similar. We do not reserve any data for validation and apply the common data augmentations of image shifting and random horizontal flipping. We find that Cutout (Devries and Taylor, 2017) further improves all CIFAR10 results by about 0.6%, but omit these statistics for conciseness.

Following ENAS, we stack $N = 5$ Normal Cells in each of the three network stages, in which all image tensors have the same shape. There are two Reduction Cells in between, each halving height and width while doubling the number of feature channels. The network parameters θ are trained with SGD over 630 epochs, the learning rate being subject to cosine decay with six warm restarts. Weight decay and drop-path (Larsson *et al.*, 2017) regularize θ to prevent overfitting.

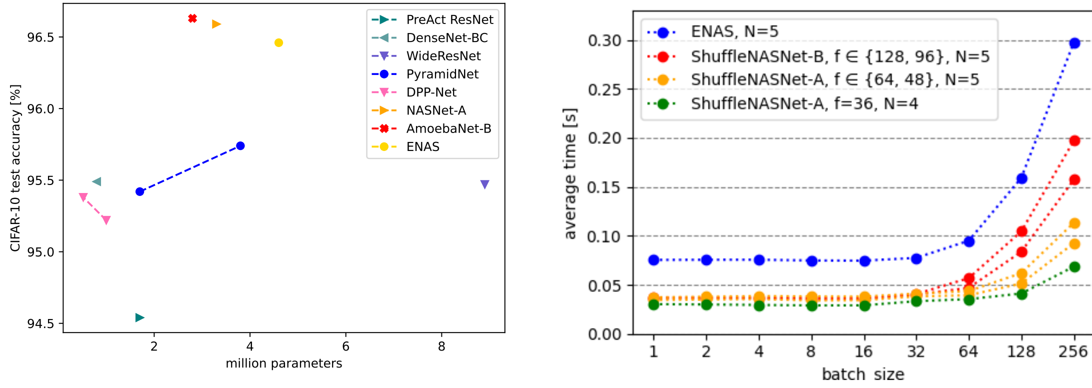
The training results can be found in Table 3.1, of which we visualize the CIFAR10 results in Figure 3.6a. We report the average test error of the last five epochs, averaged over three independent trials. By changing the number of feature channels for the first cell, we can trade a higher complexity (i.e., FLOPs and parameters) for a better accuracy, which we find to saturate at around 96.4%. ShuffleNASNet-B thus retains ENAS' performance, despite the significant reduction in cell complexity and requiring fewer parameters. Although the best networks are arguably AmoebaNet-B, NASNet-A, and PNAS-5, these methods are not economical since finding suitable architectures requires 3100, 1800, and 250 GPU hours, respectively, while ENAS and ShuffleNASNet need only around ten.

ShuffleNASNets also improve over manual designs on CIFAR100, except for the com-

Table 3.1: CIFAR results with standard regularization (flipping, shifting, weight decay, drop-path). The models in the top group are hand-crafted by human experts; those in the bottom group have been discovered automatically. ShuffleNASNets match the ENAS baseline and are vastly more parameter efficient than manually designed models with similar performance. The low-parameter models are additionally visualized in Figure 3.6a.

	Network structure		test error (%)	
Method	**layers**	**params**	**CIFAR10**	**CIFAR100**
FractalNet 2017	20	38.6M	4.60	22.85
PreAct ResNet 2016	164	1.7M	5.46	24.33
DenseNet-BC (k=12) 2017	100	0.8M	4.51	22.27
DenseNet-BC (k=40) 2017	190	25.6M	**3.46**	**17.60**
WRN-40-4 2016	40	8.9M	4.53	21.18
WRN-28-10 2016	28	36.5M	3.89	19.25
PyramidNet (α=48) 2017	110	1.7M	4.58	23.12
PyramidNet (α=84) 2017	110	3.8M	4.26	20.66

Method	**cells**	**features**	**params**	**CIFAR10**	**CIFAR100**
SMASHv1 2018			4.6M	5.53	22.07
DPP-Net-WS 2018			1.0M	4.78	
DPP-Net-M 2018			0.45M	5.84	
DPP-Net-Panacea 2018			0.52M	4.62	
NASNet-A 2018	18+2	32	3.3M	3.41	
PNAS-5 2018	9+2	36	3.2M	3.41	
AmoebaNet-B 2018	18+2	36	2.8M	**3.37**	
ENAS 2018	15+2	36	4.6M	3.54	
ShuffleNASNet-A (ours)	12+2	36	0.24M	4.93	22.92
ShuffleNASNet-A (ours)	15+2	48	0.47M	4.40	20.12
ShuffleNASNet-A (ours)	15+2	64	0.80M	4.10	19.11
ShuffleNASNet-B (ours)	15+2	96	1.79M	3.69	17.21
ShuffleNASNet-B (ours)	15+2	128	3.10M	3.57	**16.37**



(a) CIFAR10 to parameter performance of the Results in Table 3.1. Only the expensive NAS methods beat our ShuffleNASNets.

(b) ShuffleNASNets have twice the inference speed than their ENAS baseline. This can be further increased by using fewer features (channels) f.

Figure 3.6: ShuffleNASNet results in comparison.

parably huge DenseNet-BC. This is especially noteworthy since the cell designs were discovered on a similar but different data set, attesting to the transferability of network building blocks.

3.4.4 Time performance

We compare the inference speed of different ShuffleNASNet configurations with that of the ENAS baseline in Figure 3.6b, for different batch sizes between 1 and 256. The time is averaged over 1000 consecutive forward passes on a single Nvidia 1080 Ti without any data augmentation.

We find that both ShuffleNASNets are much faster than ENAS' network, which is the anticipated outcome. They feature a speedup factor of 2.00 and 2.05, respectively, which changes after batch size 32 due to what seems to be a hardware property. It is further possible to achieve a speedup factor of 2.49 by only using $N = 4$ Normal Cells per stage, at the cost of accuracy. In all cases, the networks can be executed with around 25 inferences per second, which is just enough to achieve real-time capability.

3.4.5 Variations

We experimented with minor variations to the search space and training setup to see if our initial design could be easily improved. Since the space already enables the creation of smaller cells through identity functions, explicitly varying the number of blocks (i.e., operations) B was not attempted.

Merge operations within cells: A ShuffleNASNet cell sums the results of $B = 5$ blocks (i.e., operations) unless they are already used as input in a later one. An alternative to the sum is concatenation, after which a 1×1 convolution is necessary to correct for the increased feature channel count. This approach violates rule G1 and significantly increases the parameter count but may also improve the model accuracy.

We searched for and tested several promising cells, none of which is competitive to the more straightforward initial design. The best-performing cells achieve 4.45% / 4.23% test error with 64 / 96 initial features and 1.1M / 2.4M parameters, respectively, making them strictly worse than ShuffleNASNet-A.

Bypassing reduction cells: The image tensors in reduction cells have their spatial dimensions reduced by a factor of two. The same is necessary for the tensor that bypasses the cell, making the use of a genuine identity skip connection impossible. We test three reasonable choices for the bypassing path:

- Using a factorized reduction as shown in Figure 3.4. The results of spatially shifted 1×1 convolutions are concatenated, each providing half of the required channels. While this is inexpensive and matches ENAS' and our initial design, important information may be lost.
- Two stacked 3×3 depthwise-separable convolutions, of which the first one uses a stride of 2 to reduce the spatial dimensions. This design is similar to ShuffleNet V2.
- Using the reduction cell on both channel splits.

Instead of searching for new architectures from scratch, we evaluate the three options on ShuffleNASNet-B with 96 initial feature channels. The results are in favor of our initial design, the first option. Using separable convolutions increases the parameter count by 3% and the runtime by 2%, but increases the test error by 0.15%. Applying the reduction cell on both paths worsens the performance, increasing the parameter count by 17%, the runtime by 16%, and the test error by 0.11%. Additional experiments, where the third option is already implemented in the search network rather than added late, fail to improve over ShuffleNASNet-B. Despite violating G1, the factorized reduction path is the fastest, requires the fewest parameters, and has the lowest error.

Optimal drop-path: A strong regularization technique for fragmented networks is drop-path (Larsson *et al.*, 2017), which is effectively DropOut (Srivastava *et al.*, 2014) for parallel paths. Every path has a chance to be randomly zeroed during training, regularizing the network by decreasing the dependency on specific operations. To compensate for the dropped paths, all tensors are divided by their keep chance. ENAS uses drop-path with a 50% chance, a choice that may not be optimal for our ShuffleNASNets. Since our cells are less fragmented and come in two variations, with and without BatchNorm, we tune the drop-path chance separately.

For ShuffleNASNet-B, which uses BatchNorm, the optimal probabilities match the expected 50% drop chance. In contrast, the optimal drop chance for ShuffleNASNet-A is only 0% to 10%, effectively removing the regularization technique. The effect disappears if the network is trained with additional BatchNorm, even though drop-path already adjusts the tensor magnitudes. We presume that the impact of drop-path is too powerful for the small models with less than 1M parameters unless they are additionally stabilized. Still, since they do not benefit from drop-path and BatchNorm, we use neither for our ShuffleNASNet-A results.

Auxiliary head: ENAS and other methods in the NASNet search space use an additional auxiliary head to train the final models. The idea was introduced in GoogLeNet Szegedy *et al.* (2015), which also heavily influenced the NASNet search space design. The auxiliary head is an optional structure attached at around two-thirds of the network and provides shorter paths for gradients during training.

In contrast to ENAS, we find no benefit whatsoever in using an auxiliary head and speculate that there are two reasons: Firstly, ShuffleNASNets have many skip connections within cells and through bypassing channels. Neither ResNets nor ShuffleNets require auxiliary heads, in contrast to networks from the NASNet search space with no genuine identity paths. Secondly, the auxiliary head consists of roughly 400k parameters. The comparably small ShuffleNASNets may not benefit from adding this many parameters on an optional structure that is removed later.

3.5 Conclusions

This chapter presents an alternate search space to NASNet, created by analyzing and redesigning NASNet according to ShuffleNet V2's inference speed guidelines. The resulting ShuffleNASNet space produces significantly less complex and fragmented cells than its predecessor. It contains all architectures of smaller search spaces, enabling the search with only an upper bound on complexity.

We use the economical architecture search method ENAS to find ShuffleNASNet-A and -B, two pairs of cell designs that are stacked in target-specific networks. Both models have been experimentally validated, performing much better than their human-designed competition and even comparable to the ENAS baseline. They are also structurally simpler, have fewer network parameters, and require only half of the inference time.

Chapter 4

Searching through vast architecture spaces

As seen in Chapter 3, it is possible to tailor a search space to a specific problem. Doing so ensures that the resulting architectures meet all requirements and may even be necessary to meet them, but also requires delicate changes made by domain experts. In contrast, the ideal AutoML application is set up with just the data and constraints, run without considering any details, and obtains an outstanding result. A decent and possibly infinite search space is necessary for this ambition.

This chapter explores the Prune and Replace approach, which was published in Laube and Zell (2019a) and made available on GitHub (Laube, 2019). We successfully remove some of the search space limitations by iteratively adapting the search space to the best candidate solutions, which explores increasingly specific architectures over time.

4.1 Introduction

Although architecture search methods have proven their merit by exceeding the performance of their human-designed competition in several vision tasks, they have their disadvantages. In the current state, most architecture search methods replace operations in a fixed architecture with several possible variations, resulting in a search space, and then employ a search method to select the best architectures within. Inspired by the originally human-designed network, the candidate architectures are strongly limited to fixed structures and operations. This approach generally works well since the search space is designed to contain many viable but similar solutions, which fit the data at hand.

A very different approach is the idea of evolving a network design from scratch without taking strong cues from human-designed solutions. Such methods feature a much more open search space, capable of exploring operations and network structures that human designers did not consider. Although granting more autonomy, the search space's vastly increased size and difficulty typically result in worse architectures.

The best of both worlds is necessary for NAS to truly replace human designs. While *fixed* search spaces offer efficient searches with good performance guarantees, *open*

search spaces reduce human bias, required labor, and improve task autonomy. We combine features of both sides in our Prune and Replace method, which searches for optimal candidates in a fixed search space that is periodically modified to contain additional promising candidate operations.

We implement the Prune and Replace method in DARTS (Liu *et al.*, 2019), a popular gradient-based NAS method using ENAS' search space. Our approach is faster, more robust, and discovers architectures that improve over the DARTS baseline despite considering significantly more candidate operations.

4.2 Related work

A requirement for our Prune and Replace approach is a fast and stable search method, which enables us to rank candidate operations by their usefulness. While standard hyper-parameter optimization techniques such as evolutionary algorithms or reinforcement learning enable the ranking of candidate architectures, ranking operations within such additionally requires methods such as Monte-Carlo sampling. In contrast, a gradient-based approach converges into a single result and is thus incapable of ranking different architectures but implicitly ranks all operations through their respective weightings. Therefore, we present Prune and Replace based on Differentiable Architecture Search (DARTS, Liu *et al.* (2019)), which is detailed in Section 4.2.1.

As *Prune and Replace* implies, we further require ways to remove undesired candidate operations from our search space and a way to add better ones. Sections 4.2.2 and 4.2.3 provide a brief overview of these complementary operations, network pruning and network morphisms, from an architecture search perspective.

4.2.1 Differentiable Architecture Search

Architecture search aims to find the topology and operations that maximize some objective, such as accuracy, while possibly adhering to additional objectives or constraints. The final architecture uses precisely one of the candidate operations in each place, which effectively turns NAS into a discrete optimization problem. As such, the predominant approaches originate in hyper-parameter optimization techniques like reinforcement learning or evolutionary algorithms. However, redefining the performance estimation strategy to use a single over-complete super-network (see Section 2.4.2) comes with significant benefits: Firstly, since only a single network needs to be trained, the search efficiency is tremendously increased. Secondly, all possible network operations and paths exist in parallel and can be compared to one another.

DARTS (Differentiable Architecture Search, by Liu *et al.* (2019)) relaxes the formerly discrete optimization problem to a continuous one, where the search method eventually converges into a discrete result. Since this approach does not require optimization via reinforcement learning or evolutionary algorithms, its complexity is reduced significantly.

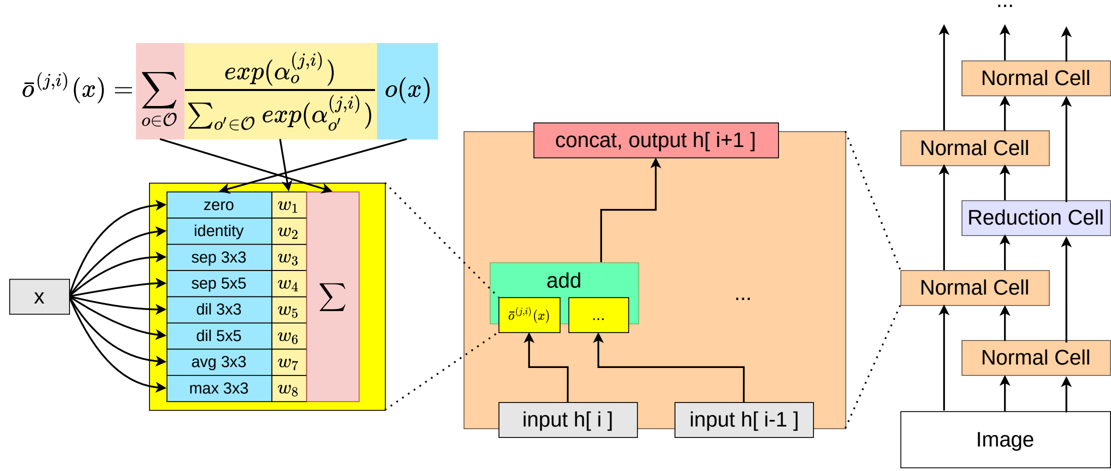


Figure 4.1: A search network based on NASNet. For conciseness, only one block and one operation from one input are displayed. DARTS learns the weighting α of the mixed substitute operation $\bar{o}^{(j,i)}(x)$, to smoothly converge into a discrete operation choice.

DARTS utilizes the reduced NASNet search space of ENAS, which is thoroughly described in Chapter 3 and partially visualized in Figure 4.1. The design of two cells has to be optimized by choosing input sources and operations in $B = 4$ blocks. The critical invention of DARTS is the relaxed optimization of operations, which can be seen in the left part of the image and is further explained below.

The final architecture will apply one operation o from the set of all candidates $\mathcal{O}$ (such as a convolution, Zero, or Max Pooling) to the current input tensor x. This categorical choice is relaxed by considering a Softmax weighting over all possible operations:

$$\bar{o}^{(j,i)}(x) = \sum_{o \in \mathcal{O}} \frac{exp(\alpha_o^{(j,i)})}{\sum_{o' \in \mathcal{O}} exp(\alpha_{o'}^{(j,i)})} o(x) \tag{4.1}$$

for an operation between a pair of computational nodes (j,i), parameterized by an architecture weight vector $\alpha^{(j,i)}$ of size $|\mathcal{O}|$. The architecture search task is thus reduced to learning a set of continuous variables $\alpha = \{\alpha^{(j,i)}\}$ that weigh the relative importance of all candidate operations $o \in \mathcal{O}$ on all edges (j,i) in the over-complete super-network. At the end of the search process, the architecture is discretized by replacing each mixed operation $\bar{o}^{(j,i)}$ with the single operation o that has the highest weighting $\alpha_o^{(j,i)}$.

As known from Equations 2.4 and 2.5, NAS is fundamentally a bi-level optimization problem: To find the optimal architecture a^*, it is required to know the optimal network weights $\theta^*(a)$ for any considered architecture $a \in \mathcal{A}$. Since an exact solution is prohibitively expensive, it is approximated by updating α and θ in turns, using single-step gradient updates:

$$\nabla\theta = \nabla_\theta \mathcal{L}(\mathcal{D}^{train}, \theta(\alpha)) \tag{4.2}$$

$$\nabla\alpha = \nabla_\alpha \mathcal{L}(\mathcal{D}^{valid}, \theta(\alpha)) \tag{4.3}$$

θ and α are updated on the disjoint training and validation data sets $\mathcal{D}^{train}$ and $\mathcal{D}^{valid}$ respectively, to improve the generalizability of the single discovered solution. The continuous architecture changes further result in a smooth and stable search phase.

However, continuous search has its drawbacks. Since the gradient computation requires computing all possible paths in every step, even a tiny proxy search network has high computational costs and memory requirements. Additionally, the gradient-based updates implicitly favor non-parameterized operations such as Skip or Pooling. DARTS imposes a hard limit on their number to avoid networks consisting of almost exclusively skip connections. Both problems have been approached in a variety of ways, in the many methods derived from DARTS. Since our method requires solving the first problem uniquely and is not prone to the second, we refrain from further details.

4.2.2 Network pruning

The removal of (nearly) unimportant weights is a widely used technique in the history of deep neural networks. By setting their values to zero, the resulting sparse matrices can be optimized to require fewer FLOPs and memory storage, often increasing inference speed and network generalizability. It is further possible to structure the pruning process to set all weights connecting unimportant neurons to zero. These neurons are effectively removed, resulting in smaller and cheaper networks.

Both pruning approaches can be seen as a subset of architecture search. Due to the improved network efficiency, structured pruning is of particular interest. Frameworks such as NetAdapt (Yang *et al.*, 2018) or Network Slimming (Liu *et al.*, 2017) iteratively remove feature channels from trained networks where they are least required and eventually reduce the model to a fraction of its former size.

Furthermore, given an over-complete super-network, finding the best-performing architecture can be viewed as a pruning task. Redundant connections and operations are removed until only the single candidate of the highest importance remains. ASAP (Anneal, Search and Prune, Noy *et al.* (2020)) use a gradient-based optimization that pragmatically prunes every candidate below a weighting threshold while steadily increasing an architecture temperature factor for faster convergence. P-DARTS (Progressive Differentiable Architecture Search, Chen *et al.* (2019)) iteratively runs the DARTS method three times. The lowest weighted candidates are removed after every iteration, which reduces the memory requirement. This makes using a larger super-network possible in the next iteration, which more accurately resembles the full-sized stand-alone networks.

4.2.3 Network morphisms

The term *network morphism* stems from mathematics, where a morphism is a function-preserving mapping from one structure to another of the same type. For neural networks, morphisms change the network in a way that keeps its function identical. The created child network thus inherits its parent's knowledge.

Wei *et al.* (2016) perform a systematic study of morphism operations on linear and non-linear network structures. An analogous work, Net2Net by Chen *et al.* (2016), studies morphism operations with the explicit goal of creating child networks that are capable of improving over their parents. Both works were published simultaneously and share many of their ideas, of which some are introduced in this Section.

Formally, a morphism operation M preserves the network F with parameters θ if

$$\forall x, \quad F(x, \theta) = M(F)(x, \theta') \tag{4.4}$$

Aside from case-specific exceptions, such a transformation M adds additional components and parameters to the network F and is therefore complementary to parameter pruning. Trivial examples of M are the addition of a bias vector initialized with zeros or a multiplicative factor of 1 anywhere in the network.

Convolution Kernels Padding the spatial components of a convolution kernel with zeros is a trivial way to retain its former function, irrespective of the prior and final sizes, which is visualized in Figure 4.2.

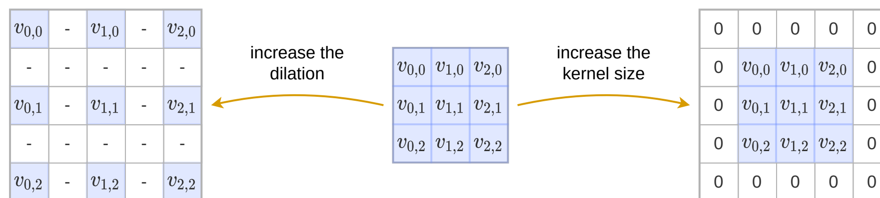


Figure 4.2: Network morphisms from a convolution kernel perspective. A 3×3 convolution (center) is turned into a 5×5 convolution (right) by padding the kernel with zeros. Increasing the dilation factor (left) only preserves the function in edge-cases (such as 1×1 convolutions) and is thus not considered a typical network morphism operation.

Inserting layers Convolution kernels (without activation function) can be initialized as an identity operation. Therefore, they can be trivially inserted at any network position, in any number, and with any spatial kernel size, without changing the network function. An example is visualized at the top of Figure 4.3. However, consecutive linear operations can be reduced to a single one. For the new layers to become useful, they must use non-linear components: activation functions.

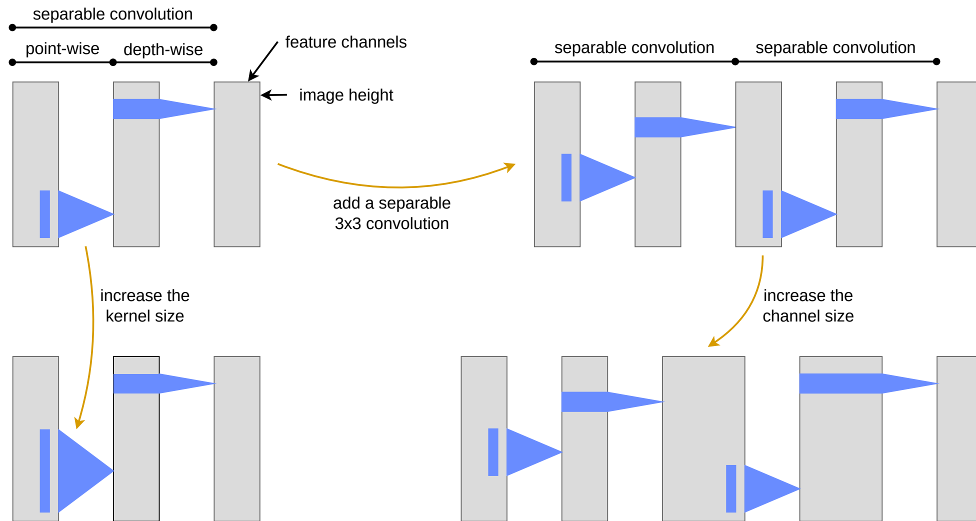


Figure 4.3: Network morphisms from a channel perspective. Every gray rectangle is an image tensor (channels $\times$ height); the convolutions operating on them are blue. A separable convolution consists of a point-wise kernel operating only on the spatial component and a depth-wise 1×1 kernel that operates only across channels. Starting in the top left, the kernel width can be expanded (bottom left), or an entirely new separable convolution can be inserted (top right). Intermediate separable convolutions can increase channel sizes (bottom right) without affecting the initial or final tensor shapes.

Inserting non-linear activation functions With few exceptions, inserting an activation function is not a network morphism:

$$ReLU(x) = ReLU(ReLU(x)) \tag{4.5}$$
$$TanH(x) \neq TanH(TanH(x)) \tag{4.6}$$

All other cases can be solved with parameterized activation functions. When interpolating between a linear and the desired non-linear activation function, simply initializing the interpolation parameter λ with approximately zero suffices:

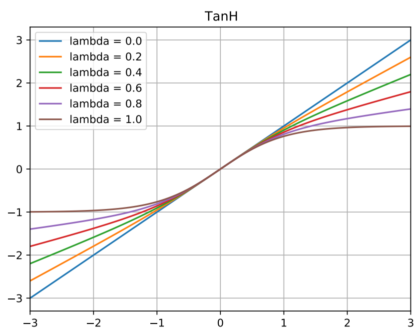


Figure 4.4: Parameterized TanH

$$PTanH(x) = (1-\lambda)\cdot x + \lambda \cdot TanH(x) \qquad \lambda \in [0,1] \tag{4.7}$$
$$TanH(x) \approx PTanH(TanH(x)) \qquad \lambda \approx 0 \tag{4.8}$$

Although inserted as a linear function, further training may gradually adjust λ to make

use of the intended non-linearity, as seen in Figure 4.4.

Widening layers The last introduced network morphism operation is the widening of existing layers, which is visualized in Figure 4.3, bottom right. Since the input and output tensor sizes are considered immutable for simplicity, two successive convolutions are necessary to enlarge only the intermediate tensor shape. The original network function is retained if one convolution is extended with random weights, the other with zeros. Experiments by Chen *et al.* (2016) also find that widening layers with random weights in all convolutions do not result in worse performance, although the child networks' learning speed decreases.

Network morphisms have been successfully applied in architecture search, often based on hill-climbing optimization. Such methods use the current parent network to generate a small number of child networks, which are trained and evaluated independently. For the next iteration, the best-performing child is selected as the new parent. Typical morphism-based architecture search thus consists of a series of locally greedy steps.

Elsken *et al.* (2018) start with a small network on CIFAR and uniform randomly insert convolution layers between randomly chosen layers and with random kernel sizes, or widen an existing layer by a random factor. Although producing results inferior to carefully designed network topologies or NAS methods, these random networks take only one GPU day to create and are on par with ResNet-18 (He *et al.*, 2016). NetAdapt (Yang *et al.*, 2018) attempts to decrease the network resource consumption by applying pruning techniques but also considers widening layers to improve the performance. Cai *et al.* (2018) use network morphisms to modify a tree-structure embedded in PyramidNet bottleneck blocks (Han *et al.*, 2017), using a meta-controller to guide the changes. The resulting networks improve over their originals in performance while requiring much fewer parameters.

4.3 Method

4.3.1 Overview

Although architecture search based on network morphisms technically searches through a near-infinite architecture configuration space, each child network evaluation necessitates further training. Given the vast possibilities, finding the optimal solution at each step is infeasible. On the other hand, NAS methods using a super-network to evaluate changes can be comparably cheap but are restricted to fixed search spaces. If the search space is fixed, it can be evaluated efficiently. If the search space is not fixed, it has a higher potential to decrease human bias and labor.

Our approach attempts a compromise: The search space in each iteration is fixed, thus enabling cheap training, evaluation, and comparisons, based on over-complete super-

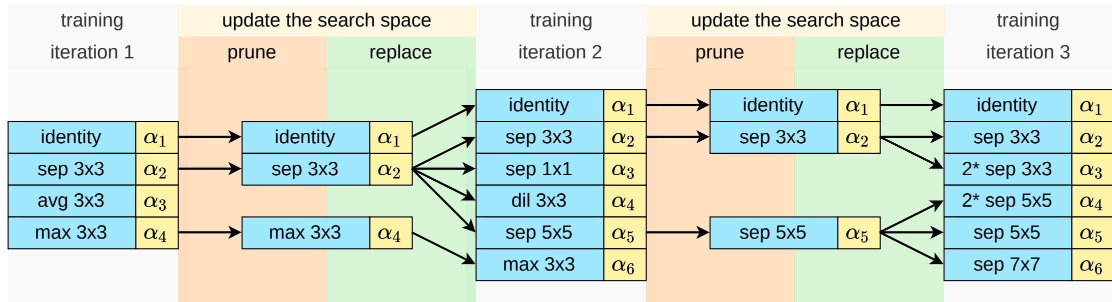


Figure 4.5: An example pool of available candidate operations over two iterations, on one edge in the super-network. The method quickly discards both Pooling operations and replaces them with variants of convolutions.

networks (see Chapter 2.4.2). The results of each iteration guide the subsequent search space changes, producing an improved fixed search space for the next iteration. Since super-networks already provide some freedom for layer connectivity, our work focuses on operations such as Identity, Pooling, and various configurations for convolutions. Our Prune and Replace process thus consists of three major steps in each iteration, which are exemplarily visualized in Figure 4.5:

1. Training the super-network with its current pool of candidate operations (gray).
2. Evaluating the super-network and pruning the worst candidates on each edge, removing their weights (orange).
3. Sampling parent operations among the survivors, weighted by their relative performance. Network morphisms are used to generate child operations, whose weights in the super-network can be initialized from their respective parent (green).

Our method thus performs locally greedy improvements of the candidate operation pool, similar to evolutionary optimization. This approach has several advantages:

- Training a super-network is much cheaper than training all the architectures it contains, even when leveraging knowledge transfer from network morphisms.
- The pool of considered candidate operations is much more comprehensive than other super-networks methods, enabling more specialized solutions. Imposing limits on allowed morphisms also introduces less human bias than providing a fixed set of operations.
- Only a limited number of candidates is considered at any time, preventing the super-network training from requiring too many resources.
- Since operations are evaluated for their usefulness over multiple iterations and in increasingly specific configurations, the results are expected to be more robust.

This iterative Prune and Replace (PR) process is implemented using the Differentiable Architecture Search (DARTS) method and thus named PR-DARTS. Future experiments

may replace DARTS in favor of more advanced operation ranking methods, which were not available at this time.

Naturally, there are also drawbacks. Our implementation limits complexity by not considering the addition of new edges between disconnected nodes in the computational graph, but it is theoretically possible to do so. Furthermore, since the available operations change for each super-network training iteration, so do the memory requirements. We solve this issue by automatically scaling the batch size, further detailed in Section 4.3.4. Another problem is the similarity of operations created from the same parent, including the parent itself, since they inherit the same network weights. We describe why that is a problem and how it can be solved in Section 4.3.3. The final drawback is the unavoidable amount of newly introduced hyper-parameters, such as the number of iterations. Since NAS experiments are expensive, our evaluation is limited to only one carefully designed set of hyper-parameters.

4.3.2 Exploring vast operation spaces with morphisms

The search space Our super-network training and evaluation methods are based on DARTS, making only the necessary search space changes enables more fair comparisons. DARTS uses a reduced variant of NASNet ((Zoph *et al.*, 2018), see Section 2.4.1): There are eight different candidate operations: Identity, Zero, Max, and Average Pooling, and four separable convolutions with kernel sizes $k \in \{3,5\}$ and dilation factors $d \in \{1,2\}$.

Since DARTS uses $B = 4$ blocks and two operations per block, the super-network contains a total of $2 \cdot (2+3+4+5) = 28$ edges in each cell, which are connected to their candidate input sources. Considering the eight operations, DARTS has 28×8 architecture weights α per cell. The search result contains only the two best combinations of input sources and applied operations per block, for a total of $2 \cdot B = 8$.

Our method applies on a per-edge basis. The 28 edges in each cell start with the same initial candidate operation pool but differ after a few steps of pruning and replacing. Some edges may immediately remove the Identity function to explore more variants of convolutions, while others will later select it as the best possible candidate.

Properties and transformations of convolutions Even though Pooling may also benefit from kernel size changes, we limit all morphism operations exclusively to separable convolutions. The initial operation pool must therefore contain all operations that can not be created later: Identity, as well as 3×3 Max and Average Pooling. The fourth and final operation is a separable convolution with four dimensions of interest:

1. The number of stacked separable convolutions n, also named *depth*. All operation candidates in the NASNet search space consist of $n = 2$ stacked separable convolutions, which is variable in our case. Our initial operation uses $n = 1$.
2. As visualized in the bottom right of Figure 4.3, given $n > 1$, the layer width $w_i \in \{w_1, ...w_n\}$ of intermediate tensors t_i may be changed.

3. The spatial kernel size, which is initialized as $k = 3$.
4. The kernel dilation factor, which is initialized as $d = 1$.

To modify the single initial convolution along the four dimensions, we use the following six transformations. Although only the first three are true network morphisms, they all allow us to initialize the weights of the generated child operations based on their respective parents.

1. Increasing the spatial kernel size by padding it with zeros, as seen in the right of Figure 4.2 and the bottom left of Figure 4.3. To limit complexity, all stacked convolutions of the same candidate always use the same kernel size.
2. Widening a layer, as seen in the bottom right of Figure 4.3. This transformation requires the candidate operation to have at least $n = 2$ stacked layers so that the final tensor shape remains unaffected. Since the stacked layers may have different widths if $n > 2$, the transformation will always enlarge the smallest widening factor w_i of an operation with factors $\{w_1, ...w_n\}$, and prefer the leading convolution if all are equally wide. The width w_i is considered a multiplicative factor to the default output channel count and is always increased by 1.
3. Inserting a layer at the end of an existing convolution candidate, increasing the depth n by 1. The new separable convolution is initialized as identity function and with width $w_n = 1$. Since the activation function is always ReLU, it is not necessary to parameterize it to retain the network function (see Section 4.2.3).
4. Decreasing the spatial kernel size k by 2, which applies to all stacked separable convolutions in one candidate operation. The child operations inherit only the spatial center weights; all outer weights are removed.
5. Increasing the kernel dilation factor d by 1, thus widening the receptive field. Although this child operation retains all weights of its parent, as shown in Figure 4.2, the network function changes.
6. Decreasing the kernel dilation factor d by 1.

With limited reversibility, this set of transformations can change a separable convolution operation along the four dimensions of interest. The spatial kernel is allowed to increase and decrease both size k and dilation d. However, the stacking n and widening w of layers can only be increased to avoid accidental performance drops by pruning relevant network structures. Furthermore, as we impose restrictions to the dimensions $\{n, w, k, d\}$, a given separable convolution can usually not be modified in all six ways.

Implementation choices The concrete algorithm implementation on GitHub (Laube, 2019) features some additional heuristics that are added to improve the efficiency:

Firstly, except when discretizing the architecture after the final iteration, at least one convolution operation must remain in each candidate pool. Doing so guarantees that the replacement of candidates via network morphisms is always possible.

Secondly, it is avoided to try any convolution variation in the same candidate pool twice. Since such an operation currently exists or was already removed, it is more promising to evaluate a different candidate that is not redundant or known to perform poorly. The only exception happens when all transformations fail to produce novel variations, in which case a previously discarded candidate may be accepted again.

4.3.3 Operation similarity

Similarity as a problem Prune and Replace is intended to gradually specialize the architecture search space by removing poorly performing operations and adding further variations of the survivors. The therefor required operation ranking is obtained from training the super-network with DARTS, which weighs all candidate operations $o \in \mathcal{O}$ on a super-network edge (j,i) with a Softmax over the architecture weights $\alpha^{(j,i)} = \{\alpha_1^{(j,i)}, ...\alpha_{|\mathcal{O}|}^{(j,i)}\}$. At the end of each algorithm iteration, the candidates with the lowest α values are removed.

Suppose the two operations o_1 and o_2 on edge (j,i) are identical in every aspect, and they start with $\alpha_1^{(j,i)} = \alpha_2^{(j,i)}$. Consequentially, their network and architecture parameters will remain identical even after training. This becomes a problem when o_1 and o_2 are ranked against a competing operation o_3, which has no such twin. Since o_1 and o_2 are identical and therefor redundant, their individual Softmax weighting is much lower than that of an equally important o_3:

$$exp(\alpha_1^{(j,i)}) + exp(\alpha_2^{(j,i)}) \approx exp(\alpha_3^{(j,i)}) \tag{4.9}$$

As a result, both o_1 and o_2 are likely removed in the next Pruning step.

While an operation pool will not contain identical candidates, newly generated ones start with inherited network weights and a very similar structure as their respective parents. Consequentially, not taking precautions first encourages a good candidate operation to father children, only to eliminate the entire family one iteration later. Even if a member survives, the lower architecture weighting significantly reduces its probability of being selected as a parent. We refer to this as the *group similarity problem*.

Given the operation ranking via DARTS, there is no elegant solution to solve the problem. Instead, a collection of design decisions attempt to mitigate the issue:

- At least one convolution candidate must survive at all times, except when finalizing the network in the last pruning step.
- Prune and replace only a few operations in each iteration. Having fewer very similar candidates reduces the sharing problem and makes removing other operations possible. To compensate, use more iterations that are individually shorter (fewer training epochs).
- Prefer pruning at most as many candidates as the previous iteration added. Even in

the worst case, at least one of the similar operations survives, which can automatically be considered the best among them.
- Finalize the architecture slowly by pruning only one candidate per operation pool in the last few iterations. Similar candidates can be safely eliminated one by one without disposing of them all at once.

Quantifying similarity Although the design decisions mentioned before do not require the quantification of similarity, it is a much-needed metric to analyze the algorithm's behavior and the changes in the search spaces.

As described in Section 4.3.1, each convolution operation has four mutable properties: how many separable convolutions are stacked n, their channel width factors $w = \{w_1, ...w_n\}$, their spatial kernel size k, and their dilation factor d. Since w already implies n, only three need to be compared. Identity is parameter-free, while Pooling operations have two parameters: kernel size and type (mean or max pooling).

We define the similarity between any two operations as 0 if they are of different types and up to 1 if their properties are the same, independent of their network weights θ. As presented in Table 4.1, every property that differs between the two operations reduces the similarity value, depending on the number of compared properties. Candidates may partially agree on a specific width w, which is also reflected in the similarity value. Since each morphism operation changes only a single property, newly generated separable convolutions have a similarity of at least 0.75 to their respective parent,

By computing the average pair-wise similarity of every candidate with all others, the concept of operation similarity can also be transferred to candidate pools. A constantly low similarity value would indicate that the group similarity problem slows or prevents the specialization of a pool, while a value that rises quickly indicates the opposite.

Table 4.1: Example similarity values between pairs of operations.

		Pooling $k=3, max$	SepConv $k=3,\ d=1,\ w=\{2,1\}$
Identity		0	0
Pooling	$k=3,\ max$	1.00	0
Pooling	$k=3,\ mean$	0.67	0
SepConv	$k=3,\ d=1,\ w=\{2,1\}$	0	1.00
SepConv	$k=3,\ d=1,\ w=\{1,1\}$	0	0.92
SepConv	$k=3,\ d=1,\ w=\{1\}$	0	0.75
SepConv	$k=3,\ d=2,\ w=\{1\}$	0	0.50
SepConv	$k=5,\ d=2,\ w=\{1\}$	0	0.25

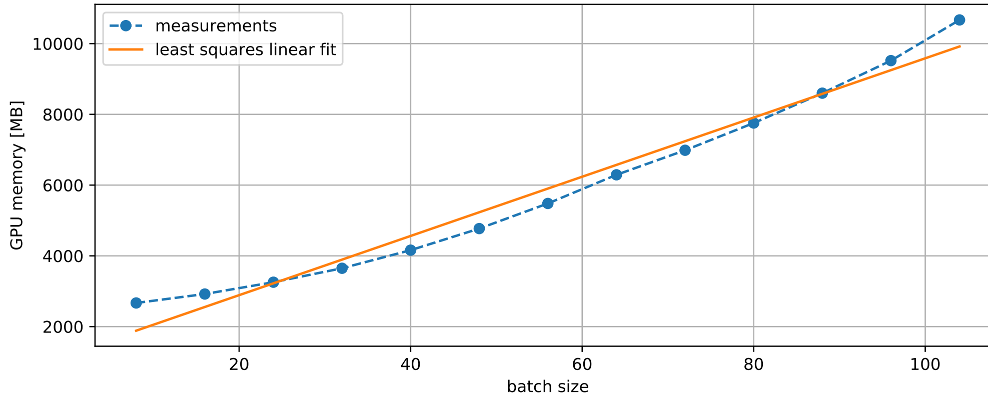


Figure 4.6: GPU memory consumption of a ResNet-18 on 224×224×3 sized images, as commonly used when training on ImageNet (Deng *et al.*, 2009).

4.3.4 Changing hardware requirements

A common choice for batch sizes is to simply fill the GPU memory, which minimizes the network training time. For a DARTS super-network with eight highly fragmented cells, an Nvidia 1080 Ti GPU with roughly 11 GB VRAM supports the training on CIFAR10 using batch size 64. Chen *et al.* find that training a larger super-network improves the search result, which strains the memory requirements even further. They progressively free GPU memory by pruning the worst performing candidate operations, thus enabling them to increase the network size.

In the case of Prune and Replace, the candidate operation pool on each network edge changes once per iteration. At first, many of the initially cheap Identity and Pooling candidates will be replaced by more expensive convolutions. The memory requirement will decrease again when the architecture is slowly finalized in the later iterations; since the candidates are then gradually removed. While a constantly small batch size would prevent out-of-memory issues at any point, using it from the start would be wasteful and slow. We therefor automatically scale the batch size during the super-network training to cope with the unknown and ever-changing requirements.

Given a minimum batch size b_{min} and a value b_{mult}, we try to find n that maximizes the batch size $b_{min} + n \cdot b_{mult}$. The term is bounded both by the maximum batch size b_{max} and the available memory, which must never be reached to prevent out-of-memory errors. As seen in Figure 4.6, the roughly linear relationship of batch size and GPU memory can be exploited to predict the expected requirements with a safety margin.

Our implementation increases the batch size every five training steps by $b_{mult} = 8$, if the linear extrapolation predicts the resulting memory consumption to stay below 95%. This approach is both simple and effective and enables training the super-network as quickly as possible at all times.

4.4 Experiments and results

In order to evaluate the Prune and Replace approach explained in this chapter, we search and evaluate architectures in three spaces of increasing size. The exact configurations are detailed in Section 4.4.1, and their results evaluated using fully sized models in Section 4.4.2. Section 4.4.3 takes a closer look at the memory requirements during the search phase, which vary depending on the currently available candidate operations. The changing candidate pools and the resulting cells are finally analyzed in Sections 4.4.4 and 4.4.5, respectively.

4.4.1 Search configuration

All hyper-parameters can be categorized into three broad groups, which are outlined separately:

1. All parameters commonly required for training neural networks, such as the number of epochs and the learning rate.
2. The parameters belonging to Prune and Replace that do not have a direct effect on the search spaces. Examples are the number of iterations or the candidate pool sizes. They are fixed in all experiments to improve comparability.
3. The parameters that restrict operation morphisms and thus define the search space limits. We vary these to experiment with different search spaces.

1. Network training parameters All hyper-parameters follow the default DARTS configuration where possible, to promote comparability of the experiments. As illustrated in Figure 4.1, a small proxy network of six normal and two reduction cells is used to optimize the architecture parameters α, which guide the Prune and Replace changes in every iteration. These weights are trained using ADAM (Kingma and Ba, 2015) with a constant learning rate of 0.0006, $\beta_1 = 0.5$, $\beta_2 = 0.999$, and a weight decay of 0.001. The super-network operation weights θ are trained using stochastic gradient descent (SGD), subject to cosine annealing in every iteration, which reduces the learning rate from 0.025 to 0.01. SGD furthermore uses a momentum term of 0.9 and a weight decay of 0.0003 to reduce overfitting.

The search network is trained on CIFAR10 (Krizhevsky *et al.*, 2009), regardless of the target data or task. CIFAR10 consists of 60,000 color images of size 32×32, of which 10,000 are reserved as a test set. The remaining 50,000 images are split evenly into a training and validation set, and used to train the network and architecture parameters, θ and α, respectively. All training images are normalized by the mean and standard deviation of the data set, randomly horizontally flipped, zero-padded to size 40×40, and randomly cropped back to 32×32.

Due to having optimization iterations, some adaptions become necessary. The SGD learning rate is reset back to its initial value at the start of each iteration, which turns

Table 4.2: The shared Prune and Replace configuration across the three experiments. Every operation pool (one on each of the 28 edges) starts with the four initial candidates and is changed a total of nine times.

	iteration									
parameter	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
total epochs		15	15	10	10	10	10	10	10	10
grace epochs		5	5	3	3	3	3	3	3	3
prune worst candidates		1	3	3	3	3	2	1	1	1
replace via morphisms		3	3	3	3	3	0	0	0	0
candidate pool size	4	6	6	6	6	6	4	3	2	1
		exploring					converging			

every individual iteration into a short DARTS version. We also train for an increased amount of total epochs due to the changing search space and automatically adjust the batch size to use the GPU efficiently under varying hardware requirements.

2. Shared Prune and Replace settings Aside from the concrete search space restrictions, the Prune and Replace method also raises the question of *how* to prune and replace.

We divide 100 training epochs for the search networks into nine iterations and apply the search space transformations at the end of each. As listed in Table 4.2, all iterations have 10 to 15 training epochs, for a total of 100. The first two iterations are slightly longer to compensate for the initially untrained network weights. Since the weights in subsequent iterations are partially trained or inherited from such parent candidate operations, ranking them requires less training time. Additionally, the first third of each iteration are so-called *grace epochs* (Chen *et al.*, 2019), during which the architecture weights are frozen. Their purpose is first to enable each candidate operation to fit the data before ranking them against each other. This primarily benefits the newly generated candidates but also all others which depend on now-pruned paths.

The design of how many operations are pruned and replaced in each of these iterations follows an exploring-converging pattern. As shown in Table 4.2, the first five iterations are used to explore the candidate space. The total number of candidates per pool is limited to six, half of which are pruned and replaced every iteration.

This approach limits the resource consumption during training and satisfies the design decisions that are theorized to mitigate the *operation similarity problem*, as described in Section 4.3.3. Since there are never more operations pruned than added, the chance to remove entire promising candidate families is reduced significantly. Following the same idea, the final four iterations are used to slowly prune operations without adding any

Table 4.3: Morphing constraints for our experiment settings. The total number of configurations includes only the obtainable configurations as described in Section 4.3.2 within five morphism steps, as well as the three initial Identity and Pooling operations. All variations of kernel size $k = 1$ combined with dilation $d = 2$ have been excluded.

	convolutions		layer expansions		
setting	**kernel sizes** k	**dilations** d	**depth** n	**width** w_i	**total**
DARTS-like (DL)	$\{3,5,7\}$	$\{1,2\}$	$\{1,2\}$	$\{1\}$	12+3
depth-restricted (DR)	$\{1,3,5,7\}$	$\{1,2\}$	$\{1,2\}$	$\{1,2,3,4,5\}$	36+3
unrestricted (UR)	$\{1,3,5,7\}$	$\{1,2\}$	$\mathbb{N}^+$	$\mathbb{N}^+$	80+3

replacements. Removing them one by one keeps at least one of the similar candidates in each operation pool until the network is finalized.

3. Search space restrictions Our experiments vary only by the restrictions we impose on transforming the kernel sizes k, dilations d, depth n, and widths w of separable convolution operations; all other parameters are kept constant. We consider three search spaces of interest and list the allowed choices for $\{k,d,n,w\}$ in Table 4.3:

- To enable a comparison with the baseline algorithm, the DARTS-like search space is restricted to operations resembling the eight candidates used by DARTS. As opposed to DARTS, we allow a kernel size of $k = 7$ and use convolutions without stacking them ($n = 1$), resulting in almost twice as many candidates.
- The restrictions of the DARTS-like search space are loosened in the depth-restricted search space, which, as the name suggests, primarily restricts the depth of the network. Contrary to before, the separable operations can now be created with kernel size $k = 1$ and a widening factor $w_i > 1$.
- The unrestricted search space imposes no restrictions on layer depth n and width w, enabling the exploration of wide and deeply stacked candidates.

As mentioned in the accompanying text of Table 4.3, the given number of operations is limited to the obtainable configurations within five morphism steps, which is the number of exploration iterations (see Table 4.2). Especially obvious in the unrestricted setting, the theoretical number of candidates satisfying the restrictions may be much larger. However, given a limited number of iterations, only a limited subset of candidates can be explored. This limitation is related to the general disadvantage of network morphism-based architecture search, where the theoretically unlimited search space suffers from non-negligible evaluation costs for every considered change.

Table 4.4: Test error on CIFAR10 and CIFAR100, all results are using standard regularization (flipping, shifting, weight decay), drop-path Larsson *et al.* (2017), and Cutout Devries and Taylor (2017). Methods that use other additional regularization techniques are excluded. We list the numbers of different discovered† and discoverable‡ operations of our normal cells, which are visualized in Figure 4.8.

		test error [%]		Search		
Method	**#params**	**CIFAR10**	**CIFAR100**	**GPU days**	**#ops**	**method**
NASNet-A Zoph *et al.* (2018)	3.3M	2.65		1800	13	RL
AmoebaNet-B Real *et al.* (2018)	2.8M	2.55		3150	19	EA
ENAS Pham *et al.* (2018)	4.6M	2.89		0.5	5	RL
ShuffleNASNet-B Laube and Zell (2019b)	3.1M	2.85	16.37	0.4	6	RL
DARTS (1st order) Liu *et al.* (2019)	2.9M	2.94		1.5	8	gradient
DARTS (2nd order) Liu *et al.* (2019)	3.4M	2.83		4.0	8	gradient
P-DARTS C10 Chen *et al.* (2019)	3.4M	2.50	16.55	0.3	8	gradient
P-DARTS C100 Chen *et al.* (2019)	3.6M	2.62	15.92	0.3	8	gradient
sharpDARTS Hundt *et al.* (2019)	3.6M	2.45		0.8		gradient
SNAS moderate Xie *et al.* (2018)	2.8M	2.85		1.5	8	gradient
NASP Yao *et al.* (2019)	3.3M	2.80		0.2	7	gradient
NASP (more operations) Yao *et al.* (2019)	7.4M	2.50		0.3	12	gradient
PR-DARTS DL1 (DARTS-like 1) (ours)	3.2M	2.74 ± 0.12	17.37 ± 0.14	0.82	$15^{\dagger}/15^{\ddagger}$	gradient
PR-DARTS DL2 (DARTS-like 2) (ours)	4.0M	2.51 ± 0.09	15.53 ± 0.29	0.82	$15^{\dagger}/15^{\ddagger}$	gradient
PR-DARTS DR (depth-restricted) (ours)	4.2M	2.55 ± 0.19	16.69 ± 0.08	0.88	$26^{\dagger}/39^{\ddagger}$	gradient
PR-DARTS UR (unrestricted) (ours)	5.4M	3.79 ± 0.24		1.10	$45^{\dagger}/83^{\ddagger}$	gradient

4.4.2 Retraining results

The discovered cells of four architecture searches have been evaluated on CIFAR10 and CIFAR100, using fully-sized networks that stack 18 normal and two reduction cells each. As seen in Table 4.4, although the primary intent of Prune and Replace is to consider vast and weakly defined search spaces, the resulting networks are on par with their competition. Since most other DARTS-derived methods attempt to mitigate the DARTS problems and improve the resulting performance, the similar test accuracy is a success for the Prune and Replace approach. The PR-DARTS UR cells from the unrestricted search space are an interesting failure in that respect, which is further investigated in Section 4.4.5.

4.4.3 Resource analysis

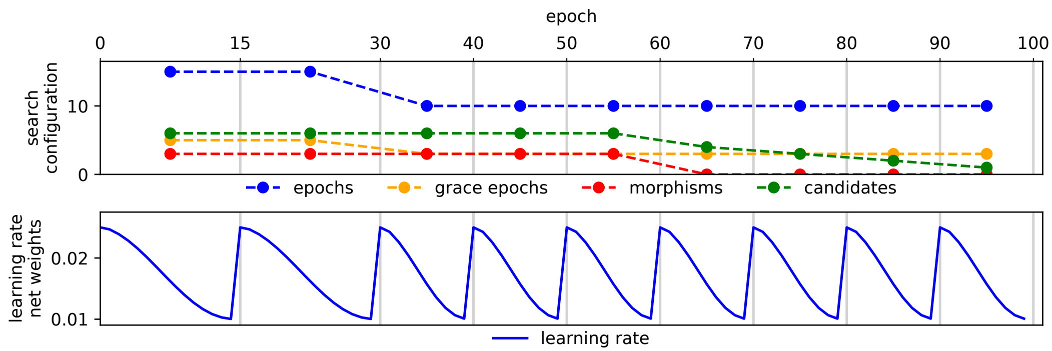
As seen in Table 4.4, PR-DARTS generally finds networks in less than one GPU day. Interestingly, this is considerably faster than DARTS, despite training the super-networks for twice as many epochs.

The first reason for this is that DARTS considers eight candidate operations in every operation pool at all times, which have to be computed in every forward pass. Our PR-DARTS method considers at most six, and even less in the last few iterations (see Table 4.2). This significantly reduces the computational costs per batch.

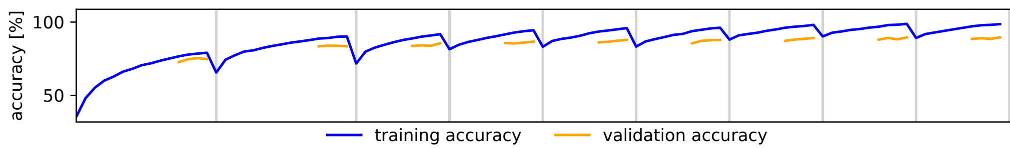
The second reason is connected to the first and can be found in Figure 4.7c: the dynamically increased batch size. While DARTS is limited to batch size 64 for the entire search process, the search experiment leading to the PR-DARTS DL2 cells maintained a larger batch size in most iterations. The batch size is continuously reduced as more variants of convolutions are added and finally increased again once operations are pruned without replacement. Exploiting the varying memory requirements may reduce the time per training epoch considerably, especially in the early and late iterations where the allowed maximum batch size of 128 is reached.

A third and final reason is the usage of grace epochs at the beginning of each iteration, during which the architecture weights α are fixed. They make up almost a third of all epochs and reduce their cost by around half.

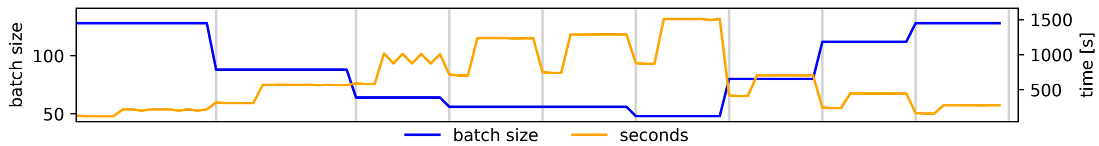
However, a search time difference of 0.7 GPU days has only a minor significance for practical matters. Evaluating the discovered cells in a full-sized network on CIFAR takes almost two days, which is still minuscule compared to ImageNet or object detection tasks. If these costs are considered, the time difference is less critical than it may appear. Nonetheless, the automatically adjusted batch size fits the goal of NAS very well, where a user can just run a search algorithm without being bothered by all the little details.



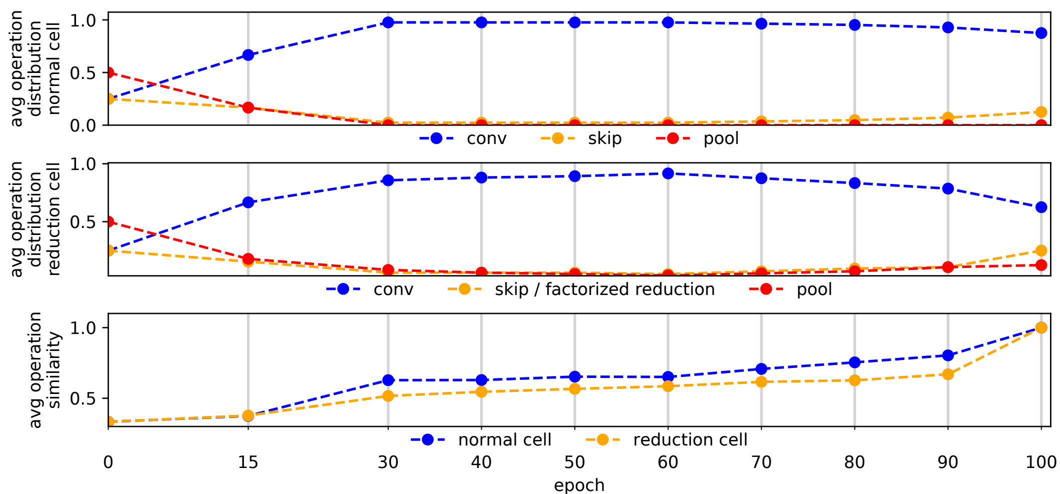
(a) The search configuration as described in Section 4.4.1.



(b) While pruning and replacing candidates has some impact on the training and validation accuracy of the search network, it improves in every iteration.



(c) As the batch size is limited by the memory requirements of the current candidate operations, the time per epoch rises.



(d) The normalized count of different operation types and their similarity averaged over all candidate pools.

Figure 4.7: Details of the cell search leading to PR-DARTS DL2, the gray vertical bars separate the nine iterations.

4.4.4 Operation similarity analysis

Contrary to all other methods in Table 4.4, Prune and Replace may consider only a subset of all theoretically possible operations. The amounts of actually discovered† and potentially discoverable‡ operations are therefore listed separately. A candidate counts as discovered if it was part of any operation pool of a normal cell, even if it was discarded immediately after.

Considering that the total number of initial and generated operations in the search iterations already exceeds 15, finding 15†/15‡ operations in the DARTS-like space is hardly surprising. The more interesting results are those of the depth-restricted and unrestricted search spaces, where only a subset of all candidates was ever considered. Since morphism transformations to separable convolutions are selected by chance and the number of discovered† operations includes all candidate pools, a much higher number was expected. The small amount of discovered† operations indicates that one configuration trait of the candidates consistently dominates the others, which are then either directly removed or unable to generate offspring.

Figure 4.7d displays the relative amounts of operation types in all candidate pools of both cell types and their similarity in every iteration. All initial candidate pools start with one Identity, two Poolings, and one 3×3 separable convolution but eventually differ as the non-renewable operations are removed in favor of convolution variations. The relative amount of convolutions only stops increasing after epoch 60, when the total number of candidates is gradually reduced. A small number of the initial Identity and Pooling operations survive all nine iterations of candidate pruning and are adopted in the final PR-DARTS DL2 cell design, as visualized in Figure 4.8b.

Interestingly, while the average operation similarity of all candidate pools in both cells is generally increasing (Figure 4.7d), this is not the case in every iteration. The similarity of the normal cells stagnates twice, which points to the *group similarity problem* (Section 4.3.3), where candidates are easily pruned when they are very similar to a competing operation. Many newly generated candidates have likely been removed immediately after, only exchanging the parent for the best child. Nonetheless, the similarity values are greater than 0.5, which means that most candidates are generally similar to one another.

After epoch 60, when the network topology converges, the similarity rises faster. Since the pools are getting smaller, the removal of a different candidate has an increased effect. At epoch 90, when only two candidates per pool remain, the average similarity in both cells is around 0.75. Most of the final two candidates thus differ by only a single property.

In summary, the *group similarity problem* slows the candidate exploration until epoch 60 but is less pronounced in the converging phase afterward. Epoch 60 is the turning point when no further candidates are added, and the pruning of existing ones is slowed down. The main reason for the stagnation is, therefore, likely the simultaneous pruning of multiple operations. Future Prune and Replace experiments should thus prune even fewer candidates per iteration, requiring more iterations to explore as many candidates as before.

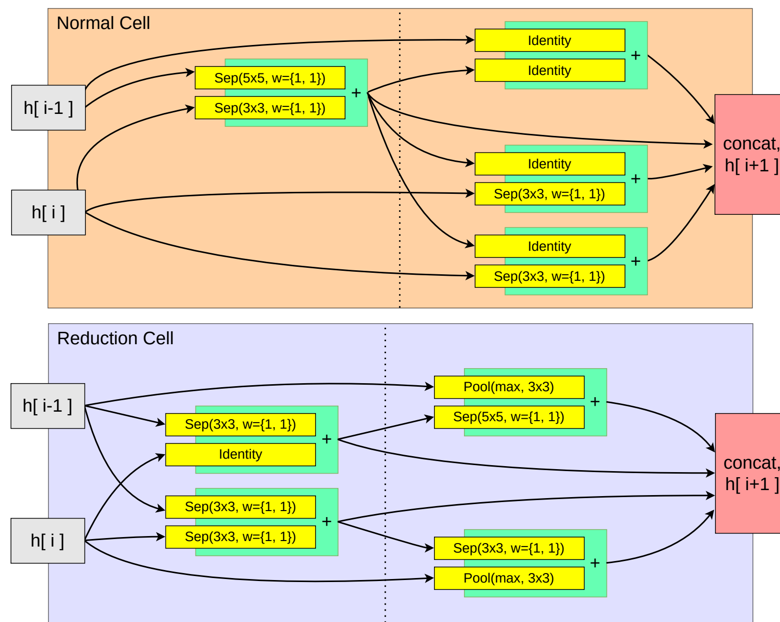
4.4.5 Cell analysis

The cells of all four different network architectures from Table 4.4 are visualized in Figures 4.8 and 4.9. Almost all cells have at least one Identity operation, while Pooling is found exclusively in reduction cells. These operations proved their usefulness by surviving nine iterations of pruning, which inadvertently solves the DARTS failure case of selecting harmfully many Identity operations. Other methods explicitly limit the number of Identity operations (Chen *et al.*, 2019), use conditional early stopping (Liang *et al.*, 2019), regularize the loss (Zela *et al.*, 2020), or carefully adjust the operation ranking formulation (Chu *et al.*, 2020a) to achieve the same effect.

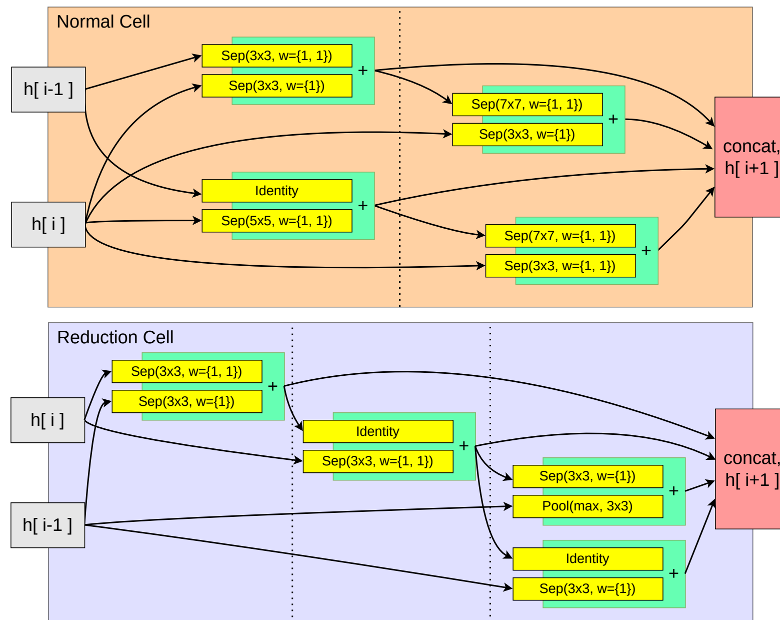
The two pairs of cells from the DARTS-like search space (Figures 4.8a and 4.8b) both prefer stacking the separable convolutions with $w = \{1,1\}$, and have only two levels of hierarchy in their normal cells. Contrary to DL1, the DL2 normal cell uses much fewer Identity operations and generally larger kernel sizes, which results in roughly 30% more network parameters and an increase in accuracy (see Table 4.4).

Once the widths of operations can be increased, as seen in Figure 4.9a, we find that half of the convolutions possess this trait. Even though the DR (depth-restricted) normal cell has many more Identity functions than DL2 (DARTS-like 2), the widened layers result in more total network parameters. Nonetheless, the significant increase in possible candidate operations was handled successfully.

However, an interesting failure case is a result of the unrestricted search space, displayed in Figure 4.9b. Although a network with these cells still competes with many manual designs, it is outclassed by its NAS competition. The UR (unrestricted) normal cell not only has three hierarchy levels, but many convolution candidates themselves are also deeply stacked. Since the cells were searched in a small proxy network of only eight cells, it seems likely that the deeply stacked convolutions were helpful to compensate for the small model size. However, when transferred to a full-sized network of twenty cells, the deep cells without skip connections become a disadvantage.

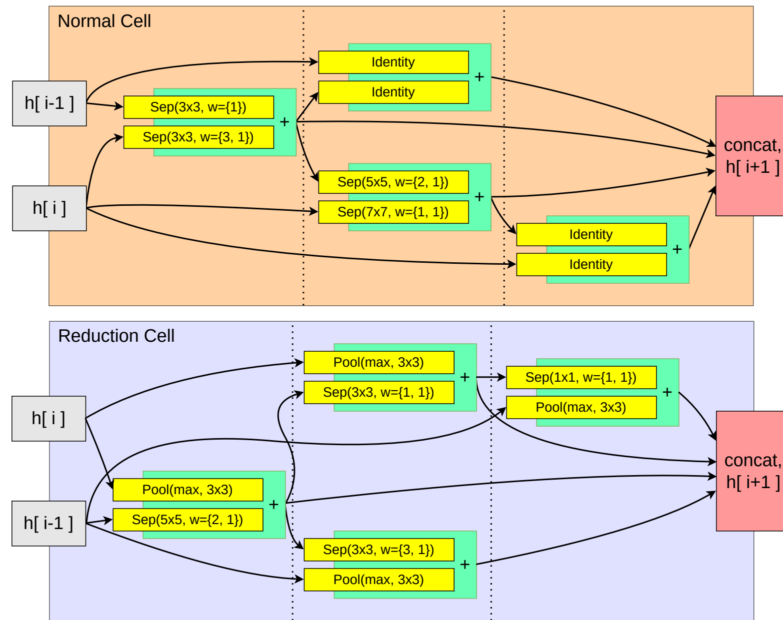


(a) DARTS-like 1 (DL1)

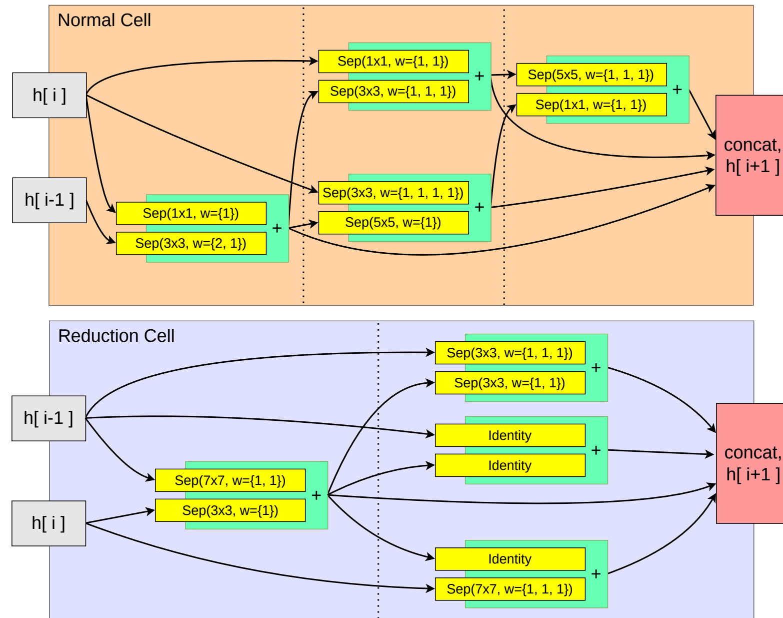


(b) DARTS-like 2 (DL2)

Figure 4.8: Normal and Reduction cells of the search results in Table 4.4. The dashed vertical lines in the cells highlight the hierarchy level of each block.



(a) Depth-restricted (DR)



(b) Unrestricted (UR)

Figure 4.9: Normal and Reduction cells of the search results in Table 4.4. The dashed vertical lines in the cells highlight the hierarchy level of each block.

4.5 Conclusions

In summary, we presented a compromise between the rigid super-network optimization and the free design changes of network morphism-based architecture search. The proposed Prune and Replace approach inherits many advantages from both sides, as it allows the efficient evaluation of the search space components and iterative improvements thereof. Applied to DARTS, the resulting PR-DARTS algorithm is faster and yields better results than its precursor while it additionally considers a more comprehensive range of candidate operations. These candidates are not defined explicitly but weakly by specifying limits to operation transformations. PR-DARTS successfully finds competitive architectures even when the width constraints are loosened, thus proving the method. However, the search for arbitrarily deep operations in small proxy networks can be considered a failure since the resulting cells can no longer be easily stacked.

The experiments are made technologically efficient by using an adaptive batch size, which predicts the GPU memory consumption from a linear extrapolation. Although simple, the training speed is maximized, and out-of-memory problems are prevented.

The major downside of Prune and Replace is complexity. There are many hyperparameters to set, and as each NAS experiment is costly, their optimal configuration is not entirely clear. Our own parameter choices were primarily guided by considerations about the *group similarity problem*, which was still affecting the cell search and reduced the exploration speed.

An interesting change in future experiments regards the candidate pools, which could be changed asynchronously and more frequently, with only minor changes each time. This should prevent the sudden pruning of many similar operations at once and improve the co-adaption of different pools. A second significant change only became possible after the original paper's publication: using a better operation ranking method than DARTS. Modern approaches provide many benefits such as being faster, requiring less GPU memory, being less susceptible to collapses, yielding better results, and being able to search directly on full-sized networks.

Chapter 5

What to expect of hardware metric predictors in NAS

In contrast to Chapters 3 and 4 that focused on search spaces and how to adapt them, the focus of this Chapter is the evaluation of hardware metric predictors. These models estimate device- and architecture-specific statistics, such as latency or energy consumption. Predictors play an important role in hardware-aware architecture search given the ever-increasing variety of network tasks and hardware platforms.

Even though predictors for hardware metrics are an important and widely used component in modern architecture search, little is known about how different models compare and affect the NAS process. This chapter is based on Laube *et al.* (2022), where these points are investigated in a large-scale study. We consider a total of 18 predictors across ten diverse datasets and find that network-based models generally perform best, although tree-based models are competitive for large amounts of training data. We then simulate the selection of architectures for different datasets, test set sizes, and degrees of predictor preciseness. These results provide insights into predictor-based NAS, what to expect from inaccurate predictors, and when they are preferable to live measurements.

5.1 Introduction and motivation

Modern neural network architectures are designed not only considering their primary objective, such as accuracy or IoU (intersection over union). Even though existing architectures can be scaled down to work with the limited available memory and computational power of, e.g., mobile phones, they are significantly outperformed by specifically designed architectures (Howard *et al.*, 2017; Sandler *et al.*, 2018; Zhang *et al.*, 2018; Ma *et al.*, 2018). Standard hardware metrics are memory usage, the number of model parameters, multiply-accumulate operations, energy consumption, latency, and more; each of which may become an optimization objective or constraint. While a low-latency objective for image classification may serve only as user convenience (objective), it becomes a necessity (constraint) in real-time person detection of autonomous cars. Additionally, the often specialized hardware may pose further restrictions on the architecture design, if e.g. the memory capacity is limited. As the range of tasks and target platforms grows,

specialized architectures and the methods to find them efficiently are gaining importance. More specifically, the efficient incorporation of hardware-specific objectives and constraints is becoming an increasingly important aspect of Neural Architecture Search.

There are two typical ways to use predictors in NAS. In the first, the search method considers all architectures separately. The candidates are sampled, evaluated (often accuracy and latency), and then ranked. An example case is presented in Chapter 3, where the network accuracy was the sole objective. Such approaches can be generalized to multiple objectives by including additional metrics as optimization targets or constraints. Instead of selecting a single architecture of maximum accuracy or minimum loss, the goal is to find the architectures that have an optimal tradeoff between the different objectives. A latency predictor is not required since directly measuring the latency of every candidate is a viable alternative. In the second usage type, a search method selects exactly one architecture from the search space, often guided by gradients. One example is DARTS, which was used to rank the candidate operations (but not architectures) in Chapter 4. However, to account for additional objectives, they must be differentiable with respect to the architecture. Ordinarily, this is not the case for latency, memory consumption, or other metrics of interest. A differentiable prediction model can provide the required gradients, not only for different architectures but also for continuous compositions thereof. In this case, a differentiable prediction model is a requisite, and live measurements are not sufficient.

Nonetheless, even if a predictor is not required, it may still be preferable. While measuring live has the advantage of not suffering from inaccurate predictions, the corresponding hardware needs to be available during the search process. Measuring live and on-demand may also significantly slow down the search process and necessitates further measurements for each new architecture search. In comparison, evaluating a predictor is effectively instant, scalable, and possible without the actual hardware, e.g., in a cloud environment. The dataset required to train a predictor has to be collected only once, while its size allows some control over the predictor's precision. Finally, changing the optimization targets simply requires the replacement of one predictor with another.

While the many advantages make predictors a popular choice of hardware-aware NAS (e.g. Cai *et al.* (2019); Xu *et al.* (2020); Wu *et al.* (2019); Wan *et al.* (2020); Dai *et al.* (2020); Nayman *et al.* (2021)), there are no guidelines on which predictors perform best, how many training samples are required, or what happens when a predictor is inaccurate. This Chapter investigates the above points by conducting large-scale experiments on ten hardware-metric datasets chosen from HW-NAS-Bench (Li *et al.*, 2021a) and TransNAS-Bench-101 (Duan *et al.*, 2021). We explore how powerful the different predictors are when using different amounts of training data and whether these results generalize across different network architecture types. To investigate the impact of inaccurate predictors, we extensively simulate the subsequent architecture selection. Our results demonstrate the effectiveness of network-based prediction models; provide insights into predictor mistakes and what to expect from them.

5.2 Related work

NAS Benchmarks Benchmarks for NAS problems are a vital component of this chapter's evaluation of predictors and their effects. An overview is provided in Chapter 2.5.1. While most benchmarks generally focus on the accuracy or loss of each architecture, TransNAS-Bench-101 (Duan *et al.*, 2021) also includes latency measurements. Furthermore, HW-NAS-Bench (Li *et al.*, 2021a) extends the popular NAS-Bench-201 benchmark (Dong and Yang, 2020) with 30 different combinations of datasets, metric, and hardware platforms. Major findings include that FLOPs and the number of parameters are a poor approximation for other metrics such as latency. Many existing NAS methods use such inadequate substitutes for their simplicity and would benefit from their replacement with better prediction models. Li *et al.* also find that hardware-specific costs do not correlate well across hardware platforms. While accounting for each device's characteristics improves the NAS results, it is also expensive. Predictors can reduce costs by requiring fewer measurements and shorter query times.

Predictors in NAS: Aside from real-time measurements (Tan *et al.*, 2019; Yang *et al.*, 2018), hardware metric estimation in NAS is commonly performed via Lookup Table (Wu *et al.*, 2019), Analytical Estimation or a Prediction Model (Dai *et al.*, 2020; Xu *et al.*, 2020). While a Lookup Table can accurately estimate hardware-agnostic metrics, such as FLOPs or the number of parameters (Cai *et al.*, 2019; Guo *et al.*, 2020; Chu *et al.*, 2019b), they may be suboptimal for device-dependent metrics. Latency and energy consumption have non-obvious factors that depend on hardware specifics such as memory, cache usage, the ability to parallelize each operation, and an interplay between different network operations. Such details can be captured with neural networks (Dai *et al.*, 2020; Mendoza and Wang, 2020; Ponomarev *et al.*, 2020; Xu *et al.*, 2020) or other specialized models (Yao *et al.*, 2018; Wess *et al.*, 2021).

Of particular interest is the correct prediction of the model loss or accuracy, possibly reducing the architecture search time by orders of magnitude (Mellor *et al.*, 2020; Wang *et al.*, 2021b; Li *et al.*, 2021b). In addition to common predictors such as Linear Regression, Random Forests (Liaw *et al.*, 2002) or Gaussian Processes (Rasmussen, 2003); specialized techniques may exploit training curve extrapolation, network weight sharing or gradient information. The experiments of this chapter follow the large-scale study of White *et al.* (2021), who compare 31 diverse accuracy prediction methods based on initialization and query time, using three NAS benchmarks.

5.3 Background

5.3.1 Multiple objectives

In the design and search for optimal neural networks architectures, it may be beneficial to incorporate additional objectives into the optimization process. While Chapters 3 and 4 compared networks by their accuracy and number of parameters, their presented methods optimize a single objective, the network accuracy.

In contrast, a multi-objective optimization method can simultaneously maximize network accuracy and minimize its latency or FLOPs. As seen in this example, the different objectives may conflict with one another. It is highly unlikely for a single network architecture to have both, the lowest latency and the highest accuracy. Indeed, the slower network architectures often have more parameters, which generally benefits their accuracy. Therefore, a multi-objective optimization method proposes the set of candidates with the optimal tradeoff, the Pareto set. For every other candidate, there exists a member in the Pareto set that is equal or better in every objective. An example is visualized in Figure 5.1.

Mathematical formulation The multi-objective problem f is composed of individual optimization problems f_i. Without loss of generality, each f_i is considered a minimization problem. Any architecture a in the search space $\mathcal{A}$ can be evaluated for k objectives, represented as a vector:

$$f_i : \mathcal{A} \rightarrow \mathbb{R}, \quad f : \mathcal{A} \rightarrow \mathbb{R}^k, \quad f(a) = (f_1(a), f_2(a), ..., f_k(a))^T, \qquad a \in \mathcal{A} \tag{5.1}$$

A solution (i.e. candidate architecture) a_1 dominates a_2 when:

$$1. \quad f_i(a_1) \leq f_i(a_2), \qquad \text{for all} \quad i \in \{1,2,...,k\} \text{ and} \tag{5.2}$$

$$2. \quad f_j(a_1) < f_j(a_2), \qquad \text{for at least one} \quad j \in \{1,2,...,k\} \tag{5.3}$$

For example, candidate a_1 dominates a_2 if it achieves a higher accuracy but at most the same FLOPs and latency. All candidates that are not dominated by any other candidate are Pareto-optimal and thus part of the Pareto set. These optimal candidates are the best possible tradeoff between the different objectives. Improving some objective f_i then always degrades at least one other objective f_j.

Comparing sets of solutions Comparing two single-objective solutions, a_1 and a_2, is easy: if $f(a_1)$ is smaller than $f(a_2)$, a_1 is the winner. Generalizing this approach to sets of solutions and multiple objectives is much harder. However, a simple and comprehensive metric does exist: the hypervolume. As visualized in the right of Figure 5.1, the hypervolume corresponds to the integral under the Pareto front with a given reference point. A large hypervolume thus corresponds to a set of solutions that, e.g., feature

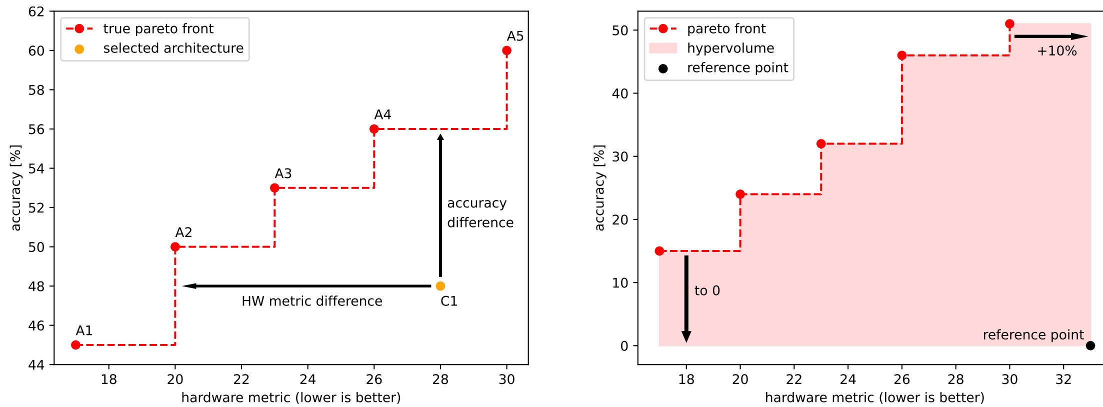


Figure 5.1: **Left:** The distances of a selected candidate architecture C1 to the Pareto front. C1 is dominated by A2, A3, and A4 of the Pareto set. A2 has a slightly higher accuracy than C1 while being much better on the hardware metric, e.g. latency. A4 has a slightly better hardware metric value, but much higher accuracy. Given several candidate architectures, their differences are averaged.
Right: We choose the reference point for the hypervolume (for two objectives: area under a Pareto front) by multiplying the highest hardware metric value from the true Pareto front with 1.1, and accuracy 0.

high accuracy and low latency values. Any unique and non-dominated solution that is removed from the evaluated set causes a reduction in this metric. For the experiments in this chapter, the reference point was dynamically chosen to be at accuracy 0, and $m = 1.1$ times the maximum hardware-metric value of the Pareto optimal architectures. This arbitrary choice is the middle ground between making the accuracy of the rightmost member of the Pareto front irrelevant ($m = 1.0$) and overemphasizing it ($m >> 1.0$).

However, the hypervolume metric has two clear weaknesses. Firstly, outliers with very high or low objective values (such as accuracy or latency) may distort the overall evaluation. This is no issue in our case, since the space is densely populated (as later seen in Figure 5.5). Secondly, hypervolume is not easily interpretable. It is not clear what a hypervolume value of 4.0 means, nor how much different other solutions are when their value is only 3.8. We therefore also compute the Mean Reduction in Accuracy (MRA), i.e., how much higher the accuracy of the Pareto set is compared to that of the currently evaluated set. If MRA is close to zero, the difference is marginal. For a single candidate C1, this is visualized in the left of Figure 5.1. Computing MRA requires the Pareto set to be known, which is infeasible in most practical problems. Still, MRA is much more intuitive than the hypervolume, since this metric clearly states how much the average network accuracy is reduced if a non-optimal set of solutions is selected. As a disadvantage, MRA makes no statements about the hardware metric, so that the absence of e.g. low-latency architectures is not reflected.

Scalarizing and differentiability As seen in Chapter 4, gradient-based NAS methods select only a single candidate architecture from the search space. Although not required in this chapter, it is possible to account for several differentiable objective functions through the means of scalarization. In the simplest form, the different objectives f_i are weighted linearly:

$$\min_{a \in \mathcal{A}} \sum_{i=1}^{k} w_i \, f_i(a) \tag{5.4}$$

By adjusting the weights $w = \{w_1, w_2, ..., w_k\}$, it becomes possible to converge to any specific Pareto optimal architecture. Cai *et al.* (2019) are the first to use this technique to find architectures with high accuracy and low latency.

The linear scalarization requires choosing w in advance, without knowing the required parameters to find solutions of particular accuracy or latency. In most cases, it is preferable to set constraints:

$$\min_{a \in \mathcal{A}} f_j(a) \qquad \text{so that} \quad f_i(a) \leq \varepsilon_i \;\; \text{for} \;\; i \in \{1, ..., k\} \setminus \{j\} \tag{5.5}$$

Doing so enables finding the architecture that maximizes accuracy, for any particular requirements. While gradient-based architecture search with constraints is challenging, it is nonetheless possible (Nayman *et al.*, 2021).

No matter how f is scalarized, gradient-based methods require every single objective function f_i to be differentiable with respect to the architecture. Chapter 4 presented DARTS, which adds architecture weights α to the competing paths in over-complete super-networks. By differentiating the loss, their gradient $\nabla\alpha$ enables the continuous optimization of the architecture. For other objectives like latency or FLOPs, an established practice is to use a differentiable prediction model. Cai *et al.* (2019) and Nayman *et al.* (2021) use a lookup table that assigns every candidate option a latency-cost. As the sum over this cost is weighted by the architecture weights α, latency becomes differentiable. Another approach is the use of neural networks (Xu *et al.*, 2020), which predict a differentiable latency for the current architecture weights α.

5.3.2 Differences between accuracy and hardware predictors

There are fundamental differences when predicting hardware metrics and the accuracy of network topologies. The most essential is the cost to obtain a helpful predictor, which may vary widely for accuracy prediction methods. While determining the test accuracy requires the costly and lengthy training of networks, measuring hardware metrics does not necessitate any network training. Consequentially, specialized accuracy-estimation methods that rely on trained networks, loss history, learning curve extrapolation, or early stopping do not apply to hardware metrics. Furthermore, so-called zero-cost proxies

that predict metrics from the gradients of a single batch are dependant on the network topology but not on the hardware the network is placed on. Therefore, the dominant hardware-metric predictor family is model-based.

Since all relevant predictors are model-based, they can be compared by their training set size. This simplifies the initialization time of a predictor as the number of prior measured architectures on which they are trained. In stark contrast, some accuracy predictors do not need any training data, while others require several partially or fully trained networks. Since an untrained network and a few batches suffice to measure a hardware-metric, the collection of such a training set is comparably inexpensive.

Additionally, hardware predictors are generally used supplementary to a one-shot network optimized for loss or accuracy. Depending on the NAS method, a fully differentiable predictor is required in order to guide the gradient-based architecture selection. Typical choices are Lookup Tables (Cai *et al.*, 2019; Nayman *et al.*, 2021) and neural networks (Xu *et al.*, 2020).

5.3.3 Model-based predictors

The goal of a predictor $f_p(a)$ is to accurately approximate the function $f(a)$, which may be, e.g., the latency of an architecture a from the search space $\mathcal{A}$. A model-based predictor is trained via supervised learning on a set $\mathcal{D}_{train}$ of datapoints $(a, f(a))$, after which it can be inexpensively queried for estimates on further architectures. The collection of the dataset and the duration of the training are referred to as *initialization time* and *training time* respectively.

The quality of such a trained predictor is generally determined by the (ranking) correlation between measurements $\{f(a)|a \in \mathcal{A}_{test}\}$ and predictions $\{f_p(a)|a \in \mathcal{A}_{test}\}$ on the unseen architectures $\mathcal{A}_{test} \subset \mathcal{A}$. Common correlation metric choices are Pearson (PCC), Spearman (SCC) and Kendall's Tau (KT) (Chu *et al.*, 2019a; Yu *et al.*, 2020b; Siems *et al.*, 2020).

5.4 Methods

Our methods follow the large-scale study of White *et al.* (2021), who compared a total of 31 accuracy prediction methods. The differences between accuracy and hardware-metric prediction, our selection of predictors, and the general training pipeline are described in this section. We then compare these predictors across different training set sizes in our experiments on HW-NAS-Bench and TransNAS-Bench-101, described in Section 5.5.

Our experiments include 18 model-based predictors from different families: Linear Regression, Ridge Regression (Saunders *et al.*, 1998), Bayesian Linear Regression (Bishop, 2007), Support Vector Machines (Cortes and Vapnik, 1995), Gaussian Process (Rasmussen, 2003), Sparse Gaussian Process (Candela and Rasmussen, 2005), Random Forests (Liaw *et al.*, 2002), XGBoost (Chen and Guestrin, 2016), NGBoost (Duan

et al., 2020), LGBoost (Ke *et al.*, 2017), BOHAMIANN (Springenberg *et al.*, 2016), BANANAS (White *et al.*, 2019), BONAS (Shi *et al.*, 2020), GCN (Wen *et al.*, 2020), small and large Multi-Layer-Perceptrons (MLP), NAO (Luo *et al.*, 2018), and a layer-operation-wise Lookup Table model. We provide further descriptions and implementation details in Appendix A.1.

Hyper-parameter tuning: The default hyperparameters of the used predictors vary significantly in their levels of hyper-parameter tuning, especially in the context of NAS. Additionally, some predictors may internally make use of cross-validation, while others do not. Following White *et al.* (2021), we attempt to level the playing field by running a cross-validation random-search over hyper-parameters each time a predictor is fit to data. Each search is limited to 5000 iterations and a total run time of 15 minutes and naturally excludes any test data. The predictor-specific parameter details are given in Appendix A.2.

Training pipeline To make a reliable comparison, we use the NASLib library (Ruchte *et al.*, 2020). Each predictor is fit 50 times on each dataset and training size, using seeds $\{0,...,49\}$.

Some predictors internally normalize the training values (subtract mean, divide by standard deviation). We choose to explicitly do this for all predictors and datasets, which reduces the dependency of hyper-parameters (e.g. learning rate) on the dataset and allows us to analyze and compare the prediction errors across datasets more effectively.

5.5 Predictor Experiments

We compare the different predictor models based on two NAS benchmarks, HW-NAS-Bench (Li *et al.*, 2021a) and TransNAS-Bench-101 (Duan *et al.*, 2021). They differ considerably by their network tasks, hardware devices, and architecture designs.

HW-NAS-Bench architecture design and datasets

In HW-NAS-Bench, each architecture is solely defined by the topology of a building block ("cell", see Section 2.4.1), which is stacked multiple times to create a complete network. Each cell is completely defined by choosing six candidate operations. Since they select from five different candidates each time, there are $5^6 = 15625$ unique cell topologies. These cells are not fully sequential but contain paths of different lengths, which is visualized in Figure 5.2.

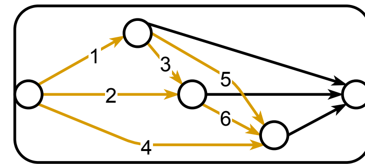


Figure 5.2: Basic NAS-Bench-201 / HW-NAS cell design. Each of the six orange paths is finalized with exactly one out of five candidate operations.

HW-NAS-Bench provides ten hardware statistics on CIFAR10, CIFAR100 (Krizhevsky *et al.*, 2009) and ImageNet16-120 (Chrabaszcz *et al.*, 2017), of which we exclude the incomplete EdgeTPU metric. Thus there are 27 data sets of varying difficulty. As detailed in Appendix A.3, 12 of them can be accurately fit with Linear Regression and only 25 training samples. Many are also very similar since their measured networks differ only by the number of image classes. We therefore select five datasets that are not trivial to learn as they are non-linear and also not redundant:

- CIFAR100, edgegpu, energy consumption
- ImageNet16-120, eyeriss, arithmetic intensity
- ImageNet16-120, raspi4, latency
- CIFAR100, pixel3, latency
- CIFAR10, edgegpu, latency

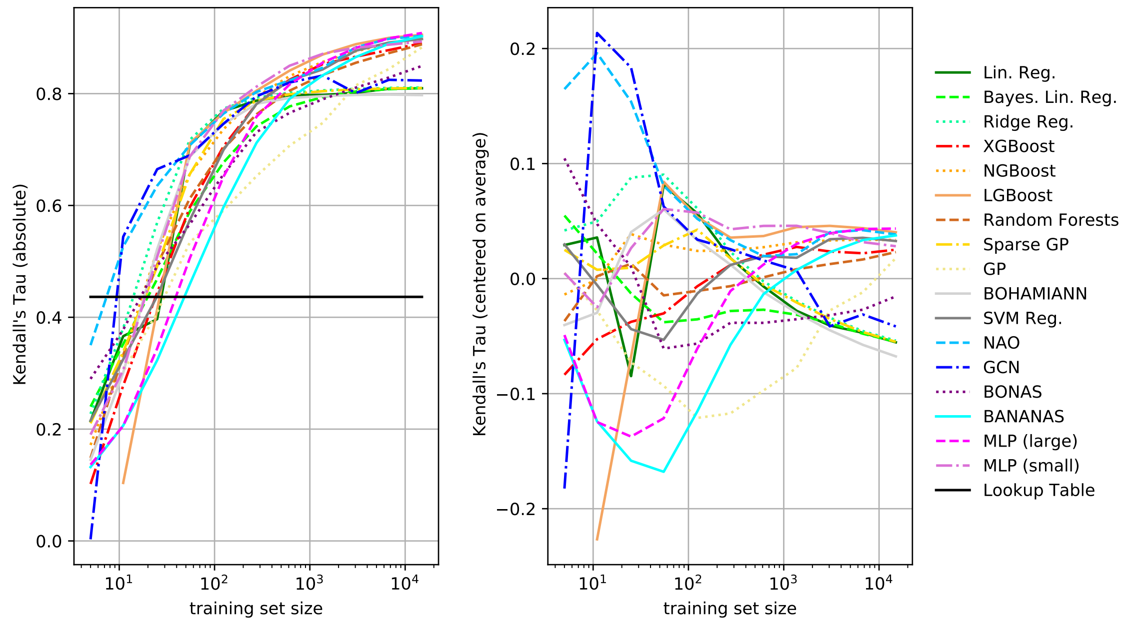
TransNAS-Bench-101 architecture design and datasets TransNAS-Bench-101 contains information for 7,352 different network architectures, used as backbones in seven diverse vision tasks. Since 4,096 are also a subset of HW-NAS-Bench, we focus on the remaining 3,256 architectures with a macro-level search space. Unlike a micro-level search space, where a cell is stacked multiple times to create a network, each network layer and block is considered individually. In particular, the TransNAS-Bench-101 networks consist of four to six pairs of ResNet blocks (He *et al.*, 2016), which may modify the image size and channels in four ways: not at all, double the channel count, halve the spatial size, and both. Every network has to double the channel count 1 to 3 times, resulting in 3,256 unique architectures. The networks may consequentially differ in their number of layers (depth), the number of channels (width), and image size at any layer.

As done for HW-NAS-Bench, we select five of the seven available datasets for their latency measurements. Aside from the self-supervised Jigsaw task, there is little difference between the cross-task latency measurements (see Appendix A.3). We evaluate the possibly redundant datasets nonetheless, since latency predictions in macro-level search spaces are an important domain for NAS on image classification and object detection tasks:

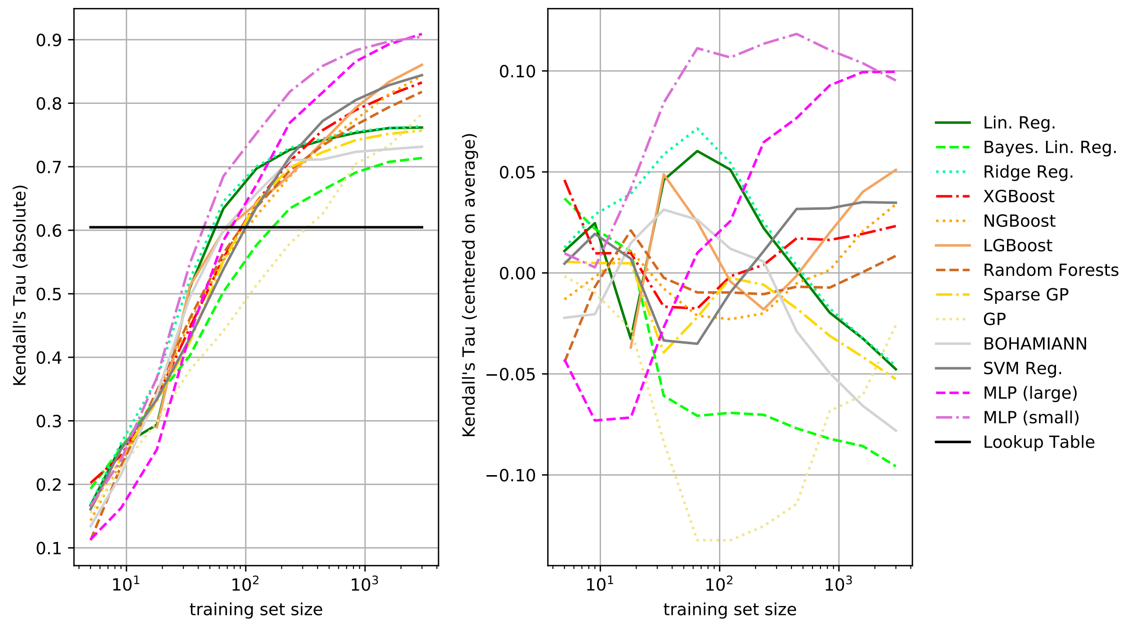
- Object classification
- Scene classification
- Room layout
- Jigsaw
- Semantic segmentation

Fitting results and comparison The results, averaged over all selected HW-NAS-Bench and TransNAS-Bench-101 datasets, are presented in Figures 5.3a and 5.3b, respectively. The left plots present the absolute predictor performance, the right ones make relative comparisons easier.

Unsurprisingly, more training samples (i.e., evaluated architectures) generally lead to better prediction results, even until the entire search space is known (aside from the test set). This is true for most of the predictors, although e.g. Gaussian Processes and BO-



(a) Results on HW-NAS-Bench. NAO performs decently at all times, and none of the prediction models requires more than 60 training samples to improve over a Lookup Table model.



(b) Results on TransNAS-Bench-101. Since all network architectures are purely sequential by design, we do not evaluate predictors that specifically encode the architecture connectivity (BANANAS, BONAS, GCN, NAO). After as few as 20 training samples, MLP models outclass all other predictors.

Figure 5.3: How well the different predictors rank the test architectures, depending on the training set size and averaged over the five selected datasets. **Left plots:** absolute Kendall's Tau ranking correlation, higher is better. **Right plots:** same as left, but centered on the predictor-average.

HAMIANN saturate early. The simple Linear Regression and Ridge Regression models also fail to make proper use of hundreds of data points but perform decently when only a few training samples are available. Interestingly, the same is true for the graph-encoding network-based predictors BONAS and GCN. While knowing how the different paths within each cell connect (see Appendix A.1) is especially useful given less than fifty training samples, the advantage disappears afterward. In contrast, the graph-encoding encoder-decoder approach of NAO performs decently at all times.

Due to their powerful rule-based approach, tree-based models perform much better given many training samples. Under such circumstances, LGBoost is a candidate for the best predictor model. Similarly, the predictions of Support Vector Machines also benefit strongly from more samples.

The model we find to perform best for most training set sizes are MLPs. They are among the top predictors at almost all times in the HW-NAS-Bench, although tree-based models are competitive given enough data. After around 3,000 training samples, thinner and deeper MLPs improve over the wider and smaller ones. The path-encoding BANANAS model behaves similarly to a regular large MLP but requires more samples to reach the same performance. This is interesting since, aside from the data encoding, BANANAS is an ensemble of three large MLP models. Even though only the first network layer is affected by the data encoding, the more complicated path-encoding proves harmful when the connectivity of the architectures in the search space is fixed. On TransNAS-Bench-101, MLP perform exceptionally well. They are much better than any other tested predictor once more than just 20 training samples are available. The small MLP model can achieve a KT correlation of 80% with just 200 training samples, which takes the best non-network-based predictor (Support Vector Machine) four times as many. They are also the only models that achieve a KT correlation of over 90%, about 5% higher than the next best model (LGBoost).

Finally, the Lookup Table models (black horizontal lines) perform poorly in comparison to any other predictor. Even though building such a model for HW-NAS-Bench datasets requires only 25 neighbored architectures, NAO and GCN perform better after just ten random samples. More than half of the predictor models require less than 25 random samples, while the worst need at most 60. On TransNAS-Bench-101, Lookup Tables perform comparably better. Building one requires only 21 neighbored architectures, and it takes most models between 50 and 100 random training samples to achieve better performance. When measured on a per dataset basis, we find that the Lookup Table models display a severe performance difference ranging from about 20% KT correlation (cifar10-edgegpu_latency and Jigsaw) to over 70% (ImageNet16-120-eyeriss_arithmetic_intensity and Semantic Segmentation, see Appendix A.3). Other models prove to be much more stable.

	HW-NAS-Bench					TransNAS-Bench-101
	Raspi4	FPGA	Eyeriss	Pixel3	EdgeGPU	Tesla V100
latency	0.45 (0.75)	0.99 (0.97)	0.99 (0.96)	0.49 (0.78)	0.21 (0.79)	0.60 (0.70)
energy		0.99 (0.97)	1.00 (0.99)		0.23 (0.79)	
arithmetic_intensity			0.84 (0.81)			

Table 5.1: The Kendall's Tau correlation of lookup tables and linear regression (in brackets, using only 124 training samples) across metrics and devices. Lookup tables perform only marginally better on the FPGA and Eyeriss devices, but considerably worse in all other cases. More detailed statistics are available in Appendix A.3.

Devices and Metrics The previously described results are based on a specific selection of HW-NAS-Bench and TransNAS-Bench-101 datasets that are hard to fit for Lookup Table models. As shown in Table 5.1, that is not always the case. The FPGA and Eyeriss hardware devices are very suitable for Lookup Tables, achieving an almost perfect ranking correlation is possible. Nonetheless, Linear Regression requires only 124 training samples to compete even there and is significantly better in every other case. We finally observe that the difficulty of fitting predictors primarily depends on the hardware device, much more than the measured metric.

5.6 Evaluating the predictor-guided architecture selection

Although the experiments in Section 5.5 greatly assist us in selecting a predictor, it is not clear what a specific Kendall's Tau correlation implies for the subsequent architecture selection. Given a perfect hardware metric predictor (Kendall's Tau = 1.0), we can expect that an ideal architecture search process will select the architectures with the best tradeoff of accuracy and the hardware metric, i.e., the true Pareto front. On the other hand, imperfect predictions result in the selection of supposedly-best architectures that are wrongly believed to be better.

To study how hardware predictors affect NAS results, we extensively evaluate the selection of such supposedly-best architectures in simulation. This approach can evaluate any combination of predictor quality, test set size, and dataset, without the technical difficulties of obtaining actual predictor models that precisely match such requirements. Since the hardware and accuracy prediction models are usually independent and can be studied in isolation, we use ground-truth accuracy values in all cases.

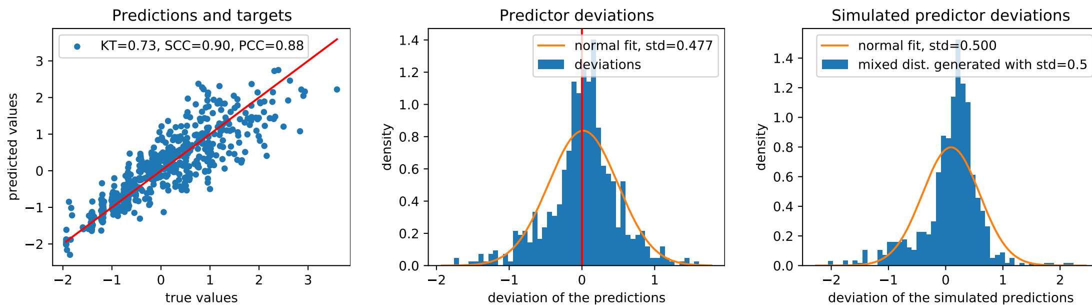


Figure 5.4: A trained XGBoost prediction model on normalized ImageNet16-120 raspi4-latency test data. **Left**: The latency prediction (y-axis) for every architecture (blue dot) is approximately correct (red line). **Center**: The same data as on the left, the distribution of deviations made by the predictor (blue) and a normal distribution fit to them (orange). **Right**: A mixed distribution can simulate typical deviation distributions as that in the center plot.

5.6.1 Simulating predictors

The main challenge of the simulation is to quickly and accurately model predictor outputs. We base our simulation on how predictor-generated values deviate from their ground-truth targets on the test set, which is explained in Figure 5.4 and further detailed in Appendix A.5. Since the simulated deviations are similar to those of actual predictors, simulated predictions are obtained by drawing random values from this deviation distribution and adding them to the ground-truth hardware measurements.

A single example of a simulation can be seen in Figure 5.5. Although most selected architectures (orange) are close to the true optimum (red Pareto front), there almost always exists an architecture that has superior accuracy and, at most, the same latency. Simply accepting the 13 selected architectures in this particular example results in a mean reduction of accuracy (MRA_{all}) of 1.06%. In other words, the average selected architecture has 1.06% lower accuracy than a comparable one on the true Pareto front. However, simply verifying the hardware metric predictions through actual measurements reveals that some selected architectures are suboptimal. By choosing only the Pareto subset of the selection, the opportunity loss can be reduced to 0.43% (MRA_{pareto}).

An important property of this approach is that it is independent of any particular optimization method. The supposedly-best architectures are always correctly identified, which is an upper bound on how well Bayesian Optimization, Evolutionary Algorithms, and other approaches can perform. The exemplary MRA_{pareto} opportunity loss of 0.43% is therefore unavoidable and depends solely on the hardware metric predictor, the dataset, and the number of considered architectures.

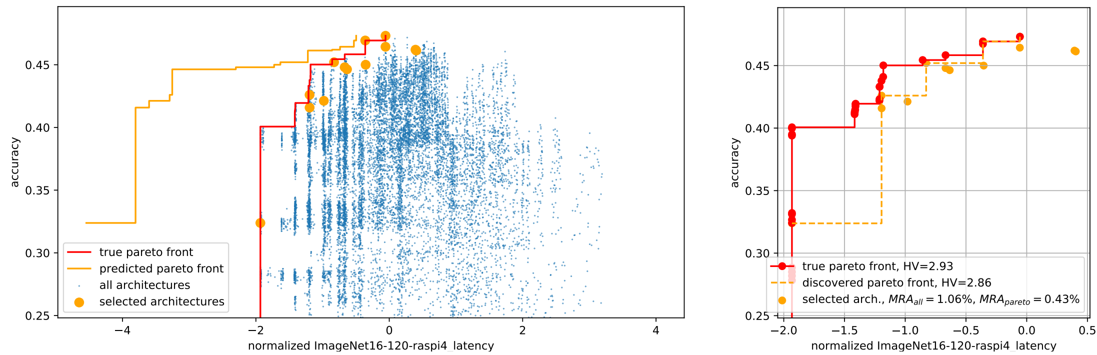


Figure 5.5: An example of predictor-guided architecture selection, std=0.5. **Left**: The simulated predictor makes an inaccurate latency prediction for each architecture (blue), resulting in the selection of the supposedly-best architectures (orange dots). Even though the predicted Pareto front (orange line) may differ significantly from the ground-truth Pareto front (red line), most selected architectures are close to optimal. **Right**: Same data as left. The true Pareto front (red) and that of the selected architectures (orange). Simply accepting all selected architectures results in a Mean Reduction of Accuracy (*MRA*) of 1.06%, while verifying the predictions and discarding inferior results improves that to 0.43%. The hypervolume (HV, area under the Pareto-fronts) is reduced by 0.07.

5.6.2 Results

We simulate 1,000 architecture selections for each of the five chosen HW-NAS-Bench datasets, six different test set sizes, and eleven distribution standard deviations between 0.0 and 1.0. As exemplarily shown in Figure 5.5, each such simulation allows us to compute the mean reduction in accuracy (MRA) and the hypervolume (HV) under the Pareto fronts. The most important insights are visualized in Figures 5.6 and 5.7, and summarized below.

Verifying the predicted results matters (Figure 5.6, left). The best prediction models achieve a KT correlation of almost 0.9, which translates to a mean reduction in accuracy of $\text{MRA}_{all} \approx 1.5\%$. That means, for each selected architecture, there exists an architecture of equal or lower latency in the true Pareto set (if latency is the hardware metric) that improves the average accuracy by 1.5%. Even though all selected architectures are believed to form a Pareto set, that is not the case. Their optimal subset has a reduction of only $\text{MRA}_{pareto} \approx 0.5\%$, a significant improvement. However, finding this optimal subset requires actually measuring the hardware metrics of the architectures selected by the used NAS method.

Furthermore, the left of Figure 5.6 aids in anticipating the MRA given a specific predictor. If one used e.g. BOHAMIANN (KT≈0.8, see Figure5.3a) instead of MLPs or LGBoost (KT≈0.9), MRA_{pareto} increases from around 0.5% to roughly 1.2%. The average accuracy of the selected architectures is thus reduced by another 0.7%, just by

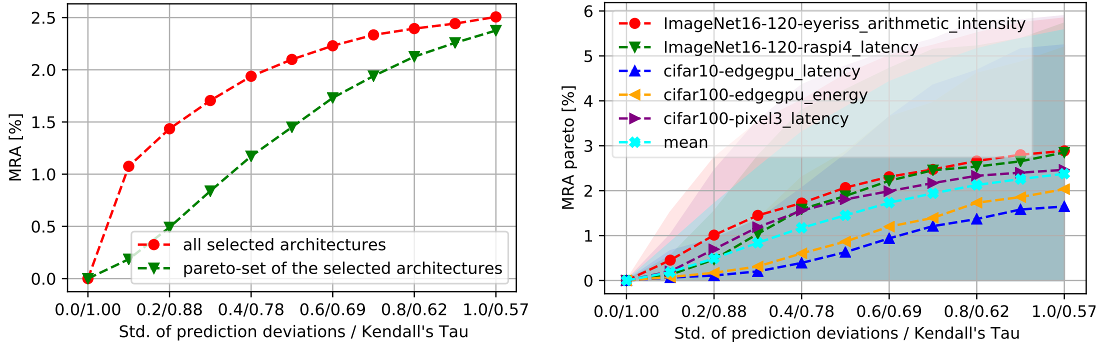


Figure 5.6: Simulation results, with the standard deviation of the predictor deviations and the resulting KT correlation on the x-axis. **Left**: Verifying the hardware predictions can significantly improve the results, even more so for better predictors. **Center**: The drops in average accuracy are dependant on the dataset and hardware metric.

using an unsuitable hardware metric predictor. Lookup Tables (KT≈0.45) are not even visualized anymore, they have an MRA_{pareto} of over 2.5%.

Another interesting observation is that the gap between MRA_{all} and MRA_{pareto} is wider for better predictors. This is a shortcoming of the MRA metric that we elaborate on in Appendix A.6.

The dataset and metric matter (Figure 5.6, right). While we generally present the results averaged over datasets, there exists some discrepancy among them. Most interestingly, predicting hardware metrics on harder classification problems (ImageNet16-120 is harder than CIFAR10) also results in a higher MRA. This is especially important since MRA is an absolute accuracy reduction. Even though the CIFAR10 networks achieve twice the accuracy of ImageNet16-120 networks, they lose less absolute accuracy to imperfect predictions. The order of MRA/dataset is primarily stable for any predictor KT correlation. Finally, as visualized by the shaded areas, the standard deviation of the MRA is generally huge. Consequentially, predictor-guided NAS is very likely to produce results of varying quality for each different predictor or search attempt, especially with less accurate predictors.

The number of considered architectures matters (Figure 5.7). We measure the hypervolume of the discovered Pareto front (i.e., the area beneath it, see Section 5.3.1), which, unlike MRA, also considers the hardware metric. Quite obviously, if the architectures from the true Pareto set are not considered, they can not be selected. To achieve the highest possible hypervolume of around 4.2 (i.e. find the true Pareto set), every architecture in the search space must be evaluated with a perfect predictor. This is impossible in most real-world cases, where only a tiny fraction of all possible architectures can ever be considered.

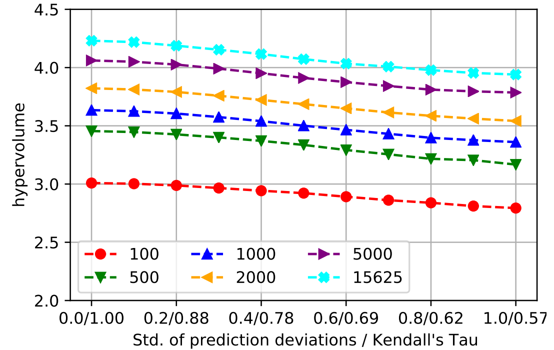


Figure 5.7: Considering more candidate architectures and using better prediction models improves the results. Larger values are better.

For HW-NAS-Bench, considering 5000 architectures with perfect live measurements and predicting the metrics for all 15625 with ranking correlation KT≈0.73 results in selecting equivalent sets of architectures. As seen in Figure 5.3a, Ridge Regression can achieve this performance with fewer than 100 training samples. Thus, a worse predictor leads to better results if it enables considering more architectures.

There is an intuitive explanation why the number of considered architectures has this effect, even for inaccurate predictions. If the predictors are perfect, sampling more architectures increases the chances of finding and identifying the (almost) optimal candidates. The same is true even if the predictors are not perfect. The example in Figure 5.5 has a small set of Pareto-optimal architectures, but many other architectures that are very close. Since candidates further away from the Pareto front are only selected if they are wrongly assigned decent predictions (e.g. for latency), having more actually-better candidates available reduces the chance for that to happen.

This insight is especially crucial for live measurements, which are accurate but slow. Similarly, estimating the network accuracy with super-networks takes much more time than predicting their performance with a neural predictor (Wen *et al.*, 2020). If the measurement of any metrics is the limiting factor, a guided selection of a cheap predictor is likely to do better.

5.7 Discussion

Chosen prediction methods Given the nature of hardware-metric prediction, only the subset of model-based predictors evaluated by White *et al.* (2021) is suitable. We extended this subset with four models, including the popular Lookup Table. We abstained from evaluating layer-wise predictors (e.g. Wess *et al.* (2021)) since such data is not available, and meta-learning predictors (Lee *et al.*, 2021) due to the vast possibilities to configure them. A separate and specialized comparison between classic and meta-learning predictors seems favorable to us.

Simulation limitations In contrast to evaluating real predictors, the simulation allows us to quickly make statements for any test set sizes and predictor-inaccuracies. However, naturally, the results are only approximations. While they match actual values, they are generally slightly pessimistic (see Appendix A.7). We also limit the simulation to HW-NAS-Bench since the changes to classification results are more accessible to interpretation than changes to loss values across different problem types. Finally, the current simulation approach can not investigate methods that require a trained one-shot network, such as gradient-based approaches. Including such methods is an interesting direction for future research.

Transferability of the results Our evaluation includes five challenging and diverse datasets based on the micro-level search space of HW-NAS-Bench and five latency-based datasets of various macro-level search space architectures in TransNAS-Bench-101. Nonetheless, we find shared trends: All tested prediction models improve over Lookup Tables with little amounts of training data. Furthermore, most predictors benefit from more training data, even until the entire search space (aside from the test set) is known. We also find that network-based predictors are generally best but may be challenged by tree-based predictors if enough training‘ data is available. Given only a few samples, Ridge Regression performs better than most other models.

Recommendations While Lookup Tables are a cheap, simple, and popular model in gradient-based architecture selection, we find a significant variance in performance across tasks and devices (see Table 5.1 and Appendix A.3). We recommend replacing such models with either MLPs or Ridge Regression, which are more stable, fully differentiable, and often take less than 100 training samples to achieve better results.

For most realistic scenarios where more than 100 training samples are available, MLP models are the most promising. They are among the top predictors on HW-NAS-Bench and demonstrate outstanding performance on the TransNAS-Bench-101 datasets. We found that specialized architecture encodings are primarily beneficial for little training data but suspect that they enjoy an additional advantage when network topologies are more complex and diverse (White *et al.*, 2021).

While the query time for all predictors is less than 0.05s and thus negligible, there is a notable difference in training time (see Appendix A.4), primarily due to the hyperparameter optimization. We recommend Ridge Regression for tiny amounts of training data and LGBoost otherwise if that is an essential factor.

If a NAS method selects architectures based on hardware metric predictions, we strongly suggest verifying the results by measuring the true metric value afterward. Doing so may eliminate inferior candidates and improve the average result substantially. Finally, if the limiting factor to a NAS method is the slow measurement of hardware metrics, using a much faster predictor may lead to an improvement, even if the prediction model is less accurate.

5.8 Conclusions

This work evaluated various hardware-metric prediction models on ten problems of different metrics, devices, and network architecture types. Our results emphasize the superiority of network-based predictors and act as a baseline for future works. We then simulated the selection process for different test set sizes and predictor inaccuracies to improve our understanding of predictor-based architecture selection. We find that the difficulty of fitting predictors primarily depends on the hardware device. Even imperfect predictors may improve NAS results if their low query time enables considering more candidate architectures. Furthermore, verifying the predictions for the selected candidates can lead to a drastic improvement in their average performance.

Finally, there are multiple interesting avenues for future research. Firstly, meta-learning hardware predictors are becoming increasingly powerful and popular but were excluded from the study due to their various configuration possibilities. A dedicated study could detail how they compare to regular predictors; and how much data and human effort are required. Secondly, the simulation currently provides an upper bound on the architecture selection. Comparisons exist between different optimization methods, such as Bayesian Optimization or Evolutionary Algorithms, but not for this particular problem. Thirdly, gradient-based or Reinforcement Learning methods are challenging to simulate since they depend on partially trained super-networks. However, as some NAS benchmarks provide extensive training information like the epoch-wise validation accuracy, an accurate simulation may be possible.

Chapter 6

Conditional super-network weights

The fast and accurate prediction of network metrics plays a central role in the search for optimal architectures. While the various predictors in Chapter 5 are trained to predict hardware metrics like latency and energy consumption, the methods used in Chapters 3 and 4 focus solely on accuracy estimation using an over-complete super-network. The cheap and accurate estimation of an architecture's loss and accuracy is arguably the most important component for an efficient architecture search process.

As presented in Chapter 5, it is generally preferable to replace slow and accurate measurements with fast but inaccurate predictors if doing so enables the evaluation of many more architectures. This is precisely the point of a super-network as used in Chapter 3. While its training costs are comparable to that of a single architecture, the capability to cheaply estimate the accuracy of any contained architecture reduces the search costs by orders of magnitude. This Chapter presents *conditional super-network weights* (Laube and Zell, 2021b), an attempt to improve the super-network's predictions through specialized network weights. Each of these weights allows candidates of different network paths to specialize towards each other, improving the super-network's estimates of the individual architectures.

6.1 Introduction

Ever since the super-network approach was first introduced by Pham *et al.* (2018), it has become the central component of many architecture search methods. By replacing the costly training of hundreds of candidate networks with the performance prediction using an over-complete network (see Chapter 2.4.2), NAS can be run in a reasonable time, even on a single GPU. The state-of-the-art methods use a variety of search strategies, such as reinforcement learning (Cai *et al.*, 2019), evolutionary algorithms (Guo *et al.*, 2020; Chu *et al.*, 2019b), and bayesian optimization (Shi *et al.*, 2019; White *et al.*, 2019). Gradient-based approaches (Liu *et al.*, 2019; Xie *et al.*, 2018; Dong and Yang, 2019; Stamoulis *et al.*, 2019; Nayman *et al.*, 2021) are especially popular since they do not require an additional optimization method. However, these methods assume that the super-network-based predictions are generally correct for any queried architecture. Consequentially, a growing research trend focuses on deepening the understanding of the

super-network training, and evaluation processes (Sciuto *et al.*, 2019; Yu *et al.*, 2020c; Chu *et al.*, 2019a; Li *et al.*, 2020; Peng *et al.*, 2020b; Nayman *et al.*, 2021). An improvement to this prediction component is also likely to improve the quality of any NAS method using it. Popular representatives include EfficientNets (Tan and Le, 2019, 2021), FBNet V3 (Dai *et al.*, 2020), and HardCoRe-NAS (Nayman *et al.*, 2021), which already deliver state-of-the-art results in the ImageNet classification and object detection challenges (Tan *et al.*, 2020). This chapter investigates a novel super-network improvement attempt based on conditional network weights.

6.2 Foundations and Related work

Super-networks This paragraph extends Chapter 2.4.2 to improve the intuition about the over-complete super-networks. An example of such a super-network is visualized in Figure 6.1. The colored arrows indicate three graph subsets that constitute different candidate architectures and share network weights by using the same operations (graph nodes).

Any graph subset that contains exactly one node per layer is a conceivable candidate network. Since the number and order of layers and their candidate operations of this example can be clearly defined, any network can be compactly described with only the indices of its graph nodes. In this case, Networks 2 and 3 can be uniquely identified with the descriptions $Net_{(2,3,3)}$ and $Net_{(2,4,3)}$.

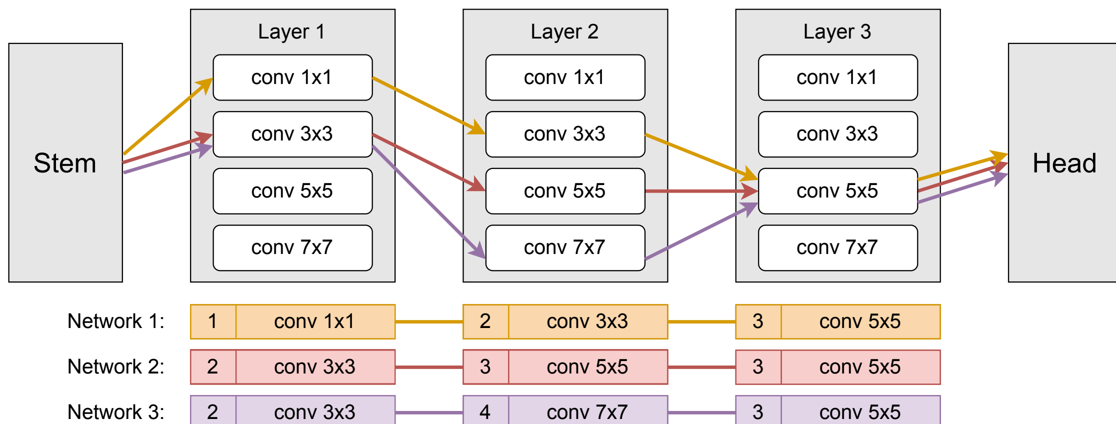


Figure 6.1: A small sequential super-network with three layers and four candidate operations in each. The connecting arrows constitute three specific architectures in the search space, subsets of the over-complete computational graph. All of them use the *conv* $5{\times}5$ operation in the last layer and thus share its weights. Aside from Layer 2, Networks 2 and 3 are identical.

Single-Path One-Shot While ENAS (Pham *et al.* (2018), used in Chapter 3) changed the currently used architecture in a super-network only once per epoch, the Single-Path One-Shot method (SPOS, Guo *et al.* (2020)) samples a new network topology for every single training batch. In both methods, every sampled architecture corresponds to a possible candidate solution, as exemplary visualized in Figure 6.1. Since the network weights of the candidates are shared, the training of each particular candidate also affects many other possible candidates in the same super-network. In contrast, gradient-based methods consider a continuous and differentiable mixture of topologies (as seen in Chapter 4) where only the converged result corresponds to a candidate solution.

While previous methods use evolutionary algorithms or other optimization techniques to guide the candidate sampling even during the super-network training, SPOS uses a simpler approach: uniform sampling. For every training batch, candidate operations are picked at random. For example, three consecutive batches could use the candidate networks $Net_{(1,2,2)}$, $Net_{(2,4,1)}$, and finally $Net_{(2,3,4)}$. Thus trained, the super-network is viewed as an unbiased collection of individually trained architectures.

The actual architecture search is performed after the super-network training. In this decoupled step, the super-network is used akin to a prediction model. Any particular architecture can be extracted from the super-network and estimated on a separate validation dataset without any additional training. Aside from training the super-network just once, evaluating an architecture thus requires only a few batches. Given that each candidate architecture can be represented with a small number of integers (e.g., $Net_{(1,2,2)}$) and evaluated cheaply, proven hyper-parameter optimization techniques for discrete values are widely used. Guo *et al.* (2020) use a simple evolutionary algorithm, which maximizes the network accuracy under a FLOPs constraint.

Adaptions to Single-Path One-Shot While the adaptions and improvements to SPOS in this paragraph are not required to understand the proposed work, they provide helpful context to the importance and current state of the field. While a distinct categorization of the various lines of work is difficult, special attention should be paid to the design of the super-networks, their training, and the subsequent search process.

Chu *et al.* (2019a) find an improved performance by sampling candidate operations in a strictly fair way, instead of uniformly random. By replacing SPOS' naive evolutionary algorithm with NSGA-II (Deb *et al.*, 2002), a sophisticated method for multi-objective optimization, Chu *et al.* (2019b) find many promising candidates with an optimal accuracy/FLOPs trade-off in a single search. Using an external teacher network, Li *et al.* (2020) obtain target outputs of intermediate layers that provide a block-wise super-network loss. Research teams from Google, Microsoft, and Facebook (Yu *et al.*, 2020a; Peng *et al.*, 2020b; Wang *et al.*, 2021a) identify specific architectures during the super-network training, which receive extra attention. Their methods are related to the Once-For-All (OFA) approach by Cai *et al.* (2020), which ranks a vast number of candidate architectures by several criteria. Trained constraint- and device-specific networks can be extracted from

OFA's super-network through the generated ranking table. This approach enables Google to supply optimized detection networks for many different Android smartphones, even though only a single super-network needs to be trained with an OFA-like method. With a novel training approach, Nayman *et al.* (2021) introduced a gradient-based architecture selection with hard latency constraints. Their approach is also based on OFA, so that specialized architectures can be extracted and finetuned in only a few hours. Lu *et al.* (2020) further demonstrate that the OFA's super-networks can be transferred and finetuned to new datasets, where they achieve exceptional performance.

Nonetheless, the changes mentioned above apply to the search for architectures and the training setup of the super-network, but not its design. Chu *et al.* (2020b) introduce *shadow batch normalization* layers that make multi-path architectures possible (as opposed to *Single-Path* One-Shot). Zhao *et al.* (2020) explore few-shot NAS (as opposed to Single-Path *One-Shot*), where a trained super-network is finetuned in different subsets of the original search space. A space that initially contained five operations can be fragmented into five different space subsets by pruning any single candidate operation. Chu *et al.* (2019b) stabilize the super-network training by adding linear 1×1 convolution to all skip-operation-candidates. This improves the gradient weighting across layers so that training candidate networks with different effective depths becomes possible.

6.3 Method

6.3.1 The problem

The super-network serves as a cheap evaluation model substituting all stand-alone networks in its search space to reduce their immense combined training costs. As seen in Figure 6.1, its weights are shared by the different candidate architectures. In particular, the three example networks use the 5×5 convolution in layer 2. However, does it make sense to use the same weights for this operation, no matter what comes before or after?

Formally, denote $O_{(2,3)}$ as the third candidate operation (5×5 convolution) in layer 2. The example network 3, $Net_{(2,4,3)}$, is thus uniquely defined by the set of its candidates $\{O_{(1,2)}, O_{(2,4)}, O_{(3,3)}\}$. Any $O_{(x,y)}$ uses the same weights no matter which particular candidate architecture is currently used. $O_{(3,3)}$ is thus part of all three example networks in Figure 6.1. A logical consequence is that all candidate operations in the second layer $\{O_{(2,1)}, ..., O_{(2,n)}\}$ need to produce structurally similar information, otherwise $O_{(3,3)}$ could not function properly. However, the candidates $\{O_{(2,1)}, ..., O_{(2,n)}\}$ may have different complexity and capacity. In this example, they differ only by their convolution kernel sizes. Still, for a 1×1 and a 7×7 convolution to produce similar outputs, the latter must not use most of its capacity. Furthermore, every operation in the third layer $\{O_{(3,1)}, ..., O_{(3,n)}\}$ must adapt to the similar outputs of any $\{O_{(2,1)}, ..., O_{(2,n)}\}$. To summarize the problem: All candidates of any one layer must adapt to similar inputs and produce similar outputs. In the worst case, the candidates with the lowest capacity limit

the intermediate network information.

In practice, several works find a substantial performance disparity between architectures that are trained independently and their equivalent subsets in a super-network (e.g. Chu *et al.* (2020a); Peng *et al.* (2020b); Li *et al.* (2020); Zhao *et al.* (2020)). Even worse, the ranking correlation between the super-network-based predictions $\{f_p(a)|a \in \mathcal{A}_{test}\}$ and their results in stand-alone training $\{f(a)|a \in \mathcal{A}_{test}\}$ may be low (see Section 5.3.3). In such cases, a super-network is an unsatisfactory prediction tool. Nonetheless, they are a proven central component of many state-of-the-art methods such as Cai *et al.* (2020); Peng *et al.* (2020b); Wang *et al.* (2021a); Nayman *et al.* (2021).

6.3.2 Conditional super-network weights

As just described, the weight sharing of the super-network paradigm encourages questionable co-adaptions of all candidate operations in the same layers. Zhang *et al.* (2020) find that decreasing the degree of weight sharing improves the search results but substantially increases the costs. The presented *conditional super-network weights* approach the problem from a different perspective by decreasing the pressure for candidate operations to co-adapt. The fundamental idea of the approach is to give each candidate operation the ability to behave differently depending on which operation generated its input. $O_{(3,3)}$ in Figure 6.1 should be able to behave differently, depending on which $O_{(2,x)}$ is part of the current network. Such pair-wise specializations are desired between any candidates in subsequent layers $(O_{(n,x)}, O_{(n+1,y)})$.

A simple approach to achieve this specialization is by choosing the weights of every candidate operation in a topology-aware way. More precisely: every single candidate has different sets of weights, which are tied to the candidate operations in the layer before. $O_{(3,y)}$ does not depend on the previous layer, but $O_{(3,(x,y))}$ does. While the previous layer does not affect the type of operation (e.g., a 5×5 convolution), its specific weights are finetuned individually. This is visualized in Figure 6.2.

Due to the complexity of this approach, there are two significant concerns: Firstly, the number of super-network weights has effectively been multiplied by the number of candidate operations. Consequentially, given the same amount of training as a regular super-network, the weights will not adapt well. An increased training time would address this issue but is naturally undesired. Secondly, while the specialized weights of each candidate (e.g., $\{O_{(3,(A,C))},...,O_{(3,(E,C))}\}$) are not supposed to be identical, it is unlikely that they should be vastly different. However, if they are trained entirely independently, this is likely to be the case. Both concerns can be addressed with a simple approach that we call *weight splitting*: Until a specific training epoch, say at three-quarters of the allocated time, the specialized weights are disabled and the super-network trained as usual. Only then are the weights of all operations $O_{(x,y)}$ *split*, i.e., copied once for each candidate operation in the previous layer. A set $\{O_{(x,(A,y))},...,O_{(x,(E,y))}\}$ is created from every candidate $O_{(x,y)}$, to enable specialization towards the prior operations A to E for the remainder of the training. Since all weight sets are initially trained as one, they are

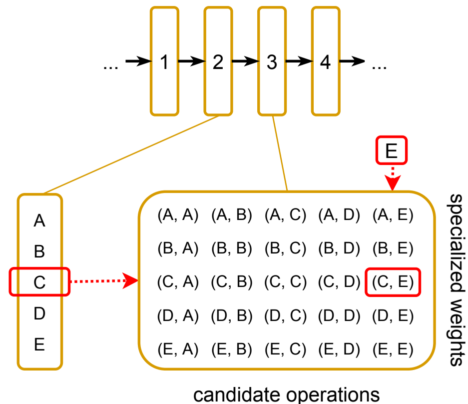


Figure 6.2: An example of conditional weights in a purely sequential super-network that has five candidate operations (A to E) in each layer. Every candidate has different weights for each candidate in the previous layer. In this particular forward pass, layer 2 is set to operation C, and layer 3 to E (marked red). The used operation in layer 3 is thus $O_{(3,(C,E))}$.

similar to one another and require no additional training time. As a disadvantage, the choice of when to split each candidate is an open question.

6.3.3 Search spaces

We evaluate super-networks modified with conditional weights in four different search spaces from the following two NAS benchmarks:

NAS-Bench-201

In the popular NAS-Bench-201 benchmark (Dong and Yang, 2020), architectures are defined by the design of a building block (cell) that is stacked multiple times to create a network. The cells differ by their chosen candidate operations, which are placed on each of the six marked edges in Figure 6.3. Thus there are $5^6 = 15625$ topologies, with paths of different lengths. We evaluate the conditional super-network weights on three search space subsets of increasing difficulty:

1. All operations are available. Finding above-average models is easy since many networks contain several zero or pooling operations and thus perform poorly.

2. The zero operation has been removed. The search space thus contains $4^6 = 4096$ architectures. Since most poorly-performing networks are not part of this search space, it is more difficult to find above-average ones.

3. Only the 1×1 and 3×3 convolutions remain, reducing the search space to just the $2^6 = 64$ architectures that make up most top-performers in both other search spaces. Since all candidates perform well, finding the best architectures in this subset is the most difficult.

Candidate operations:

- zero
- skip
- 1×1 convolution
- 3×3 convolution
- 3×3 average pooling

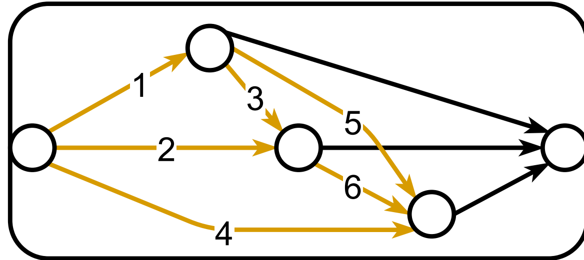


Figure 6.3: The NAS-Bench 201 candidate operations and cell design. There are three intermediate nodes and six choices among the available five operations, once for each numbered orange edge.

Implementing the added super-network weights in this search space is not as straightforward as depicted in Figure 6.2. Since there are parallel paths in a cell, choosing which other candidates to depend on is ambiguous. With respect to Figure 6.3, we have implemented the meaning of "prior" in the following way: The candidates on Paths 1, 2, and 4 have no dependency and are thus never split. Candidates on Paths 3 and 5 depend on Path 1. Those on Path 6 depend on Paths 2 and 3 and therefore split twice. The default cells have $6 \cdot 2 = 12$ candidates with operation weights (Zero, Skip, and Pooling do not have any and therefore always perform the same function). Due to the splitting, the total number of weight sets is increased to $3 \cdot 2 \cdot 5^0 + 2 \cdot 2 \cdot 5^1 + 1 \cdot 2 \cdot 5^2 = 76$. The super-network structure is detailed in Appendix B.2.

NAS-Bench-Macro

The recent NAS-Bench-Macro benchmark (Su *et al.*, 2021) lists the test accuracy of 6561 fully sequential networks evaluated on CIFAR10. Their design is inspired by the MobileNet V2 family (Sandler *et al.*, 2018), a popular starting point for modern NAS search space designs. After starting with a 3×3 convolution layer, one of three available candidates must be chosen for each of the eight subsequent layers ($3^8 = 6561$). The available candidate operations are:

- an inverted bottleneck block with kernel size 3×3 and expansion ratio 3
- an inverted bottleneck block with kernel size 5×5 and expansion ratio 6
- a skip connection

The networks have an average accuracy of roughly 90.4%, with the best network achieving 93.13%. Ordinarily, there exist $8 \cdot 2 = 16$ weight sets (skip connections do not have weights). Splitting increases that to $1 \cdot 2 \cdot 3^0 + 7 \cdot 2 \cdot 3^1 = 44$ (the weights in the first network layer do not depend on any previous layer).

6.3.4 Evaluation metrics

Even though the super-network-based predictions are often used in multi-objective optimization, we simplify the problem by considering only the network accuracy. The different objectives are generally measured or predicted independently (e.g., accuracy via super-network, latency via lookup table), so that improving them in isolation still benefits their combined application.

For a comprehensive evaluation, we first let the trained super-networks rank all architectures in the test set $\mathcal{A}_{test}$. Since detailed network statistics are known, we extend an evaluation approach that we introduced in (Laube and Zell, 2021a) and compare the mean ground-truth accuracy of the top-N selected architectures. To better understand the scale of the improvement gained from using a super-network, we also provide a normalized improvement value. It is 0 for the average and 1 for the maximum network accuracy in A_{test}. Both metrics can be seen in the experimental results in Figures 6.4 and 6.5, on the left and right y-axis, respectively. Since this form of visualization is not suitable for a detailed comparison between different experiments, the relative performance changes of the selected top-1, top-5, and top-10 networks are displayed in the right column.

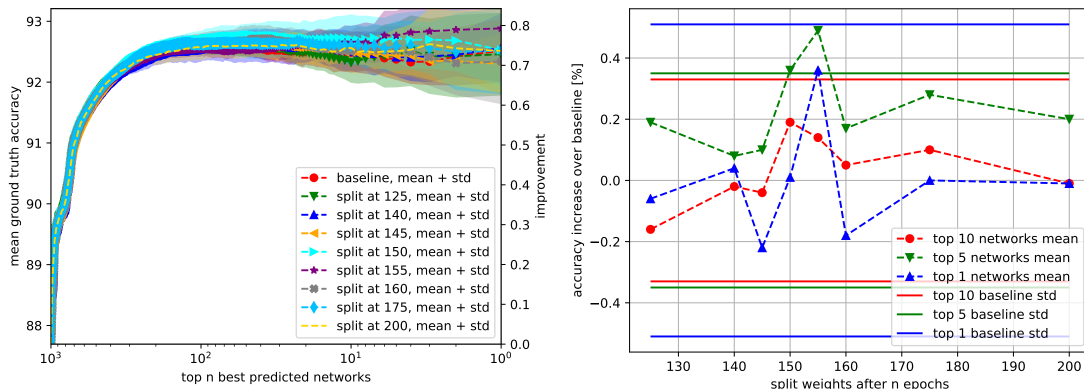
6.4 Experimental evaluation

This section evaluates the effect of adding conditional super-network weights with respect to the regular super-network baseline. Results are consistently averaged over ten independently trained super-networks, both for their architecture selections in Section 6.4.1 and the resource consumption analysis in Section 6.4.2. Their training details are listed in Appendix B.3.

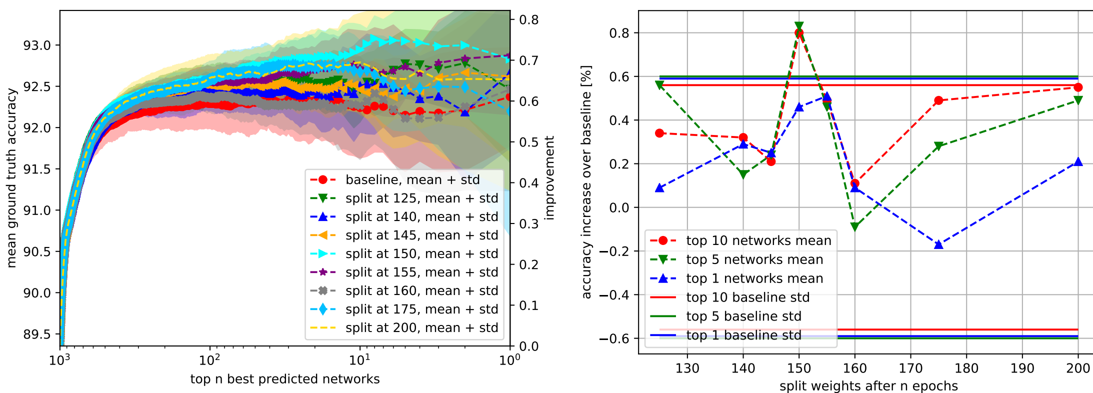
6.4.1 Search results

NAS-Bench-201 The results of splitting weights in the multi-path cell-based NAS-Bench-201 super-networks are visualized in Figure 6.4. A fascinating property that all search space subsets have in common is an improvement window when splitting at around 150 epochs of training. If the timing for weight splitting is just right, the super-networks make notably better suggestions on which networks to select, resulting in improved average accuracy.

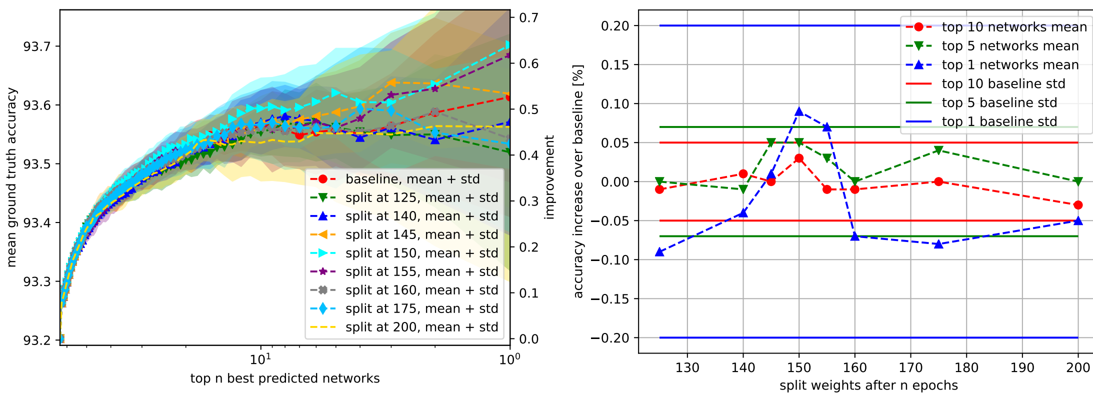
As seen in Figure 6.4a, The baseline in the full search space selects networks that achieve around 92.2% accuracy and therefore improves considerably over the space average of around 87.8%. This is also apparent in the improvement value, which is at 0.7. The selected networks are therefore already close to optimal, with little room for improvement. Nonetheless, splitting weights at around 150 epochs enhances the architecture selection further. The best value is achieved when splitting the weights after 155 epochs of training, resulting in an average improvement value of around 0.8 for the top-1



(a) All operations



(b) No Zero



(c) Only Convolutions

Figure 6.4: Visualized results of splitting weights in the NAS-Bench-201 subsets. There is a visible improvement when splitting in a narrow window of around 145 to 155 epochs of training.

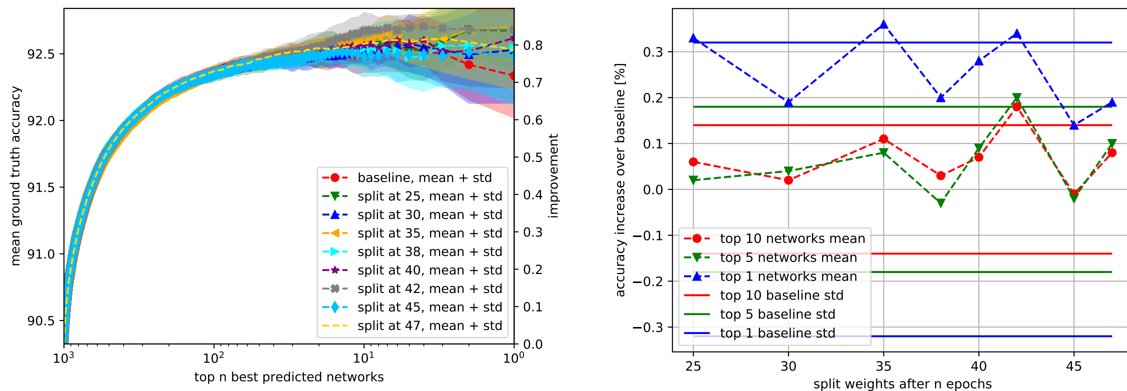


Figure 6.5: Visualized results of splitting weights in NAS-Bench-Macro. Since the baseline networks poorly select the top-1 architectures (left image, red), many split variants do better. However, there is little structured effect on the top-10 or top-5 selections.

selected architectures.

For the super-networks in the No Zero search space, this improvement peak is even more pronounced. As visualized in Figure 6.4b, the average accuracy changes of the top-5 and top-10 selections reach up to 0.8%. The selected networks are considerably better than those of the baseline if the super-network weights are split between 125 and 155 epochs of training. A notable difference to the full search space is that all super-networks have a wider standard deviation. In the absence of the Zero candidate operation, which is not present in any of the best models, the super-networks are less certain which architectures are relatively better. Nonetheless, splitting at 145 epochs of training can raise the improvement value from around 0.61 to 0.72, much closer to the optimal network performance. The average accuracy of the top-1 selected networks is improved from 92.3% to 92.8%.

Finally, the Only Convolutions search space is the only one where the top-N-selected networks never improve over the baseline by more than one standard deviation. As seen in Figure 6.4c, the reason is primarily that the baseline's standard deviation is huge. Another interesting aspect of this search space is that even random sampling results in 93.2% average network accuracy, much higher than for the other search spaces. Finding the best networks here proves more challenging. Nonetheless, an increase of the selected top-1 network improvement value from around 0.53 to 0.64 is still obtainable. The average accuracy of the selected networks increases from 93.6% to 93.7%.

NAS-Bench-Macro As seen in Figure 6.5, the effect of splitting the network weights is somewhat incomparable to any NAS-Bench-201 search space. Most notably, the top-1 selected networks are considerably better than those of the baseline at any point. To a lesser degree, the same is true for the top-5 and top-10 selected networks. The most outstanding epoch for splitting is 42, which is already close to the end of training. At this time, all top-N selections improve by more than one standard deviation over the baseline.

While the improvement peak at epoch 42 is outstanding, it is possibly not unique. However, based on only ten super-networks per splitting point, interpreting too much into the minor improvements (such as at 35 epochs) is likely premature. Further experiments are needed to verify the results in such fully sequential architecture designs and to research whether the poor performance of the selected baseline top-1 networks is purely coincidental.

Finally, the Kendall's Tau ranking correlation values across the search spaces and splitting configurations are listed in Appendix B.1. Interestingly, we observe that the ranking correlation does improve over the baseline, but only by a small amount of up to 0.05. In contrast, Figure 6.4a displays an observable peak at around 155 epochs. Splitting the super-network weights with the correct timing facilitates a consistently better selection of architectures. Nonetheless, some correlation values improved notably. In the No Zero NAS-Bench-201 search space, when computing the correlation across all architectures or just the top 50, the baseline value can be increased by around 0.05. While that does not seem like much, it is still one step towards a perfect architecture performance predictor.

6.4.2 Resource analysis

As described in Section 6.3.3, splitting increases the total number of super-network weights significantly. While the purely sequential NAS-Bench-Macro super-networks only have 2.75 times as many candidate operation weight sets as the baseline (16 to 44), this factor is 6.3 for the multi-path NAS-Bench-201 super-networks (12 to 76). As detailed in Table 6.4.2, the increase in GPU memory is marginal nonetheless, with a peak of only 1.2%. The most memory-costly component of training is the saving of intermediate tensors for backpropagation so that storing additional network weights has little effect.

Estimating changes in the training time is less reliable due to the random selection of candidate operations during training. However, a somewhat consistent trend is that NAS-Bench-201 super-networks require slightly more training time when split early. Unsurprisingly, search spaces that contain more zero-cost operations (Zero, Skip) have a shorter average training time and are thus affected more strongly by the splitting-overhead than search spaces without.

	NAS-Bench 201					
	full		no Zero		only Conv.	
	time	GPU	time	GPU	time	GPU
baseline	1862	1895	2193	1895	2783	1895
split at 125	+17.7%	+1.2%	+9.6%	+0.6%	+2.5%	+0.0%
split at 140	+14.6%	+1.2%	+9.9%	+0.6%	+2.2%	+0.0%
split at 145	+17.1%	+1.2%	+7.9%	+0.6%	+2.6%	+0.0%
split at 150	+13.8%	+1.2%	+8.1%	+0.6%	+3.0%	+0.0%
split at 155	+14.8%	+1.2%	+8.9%	+0.6%	+2.1%	+0.0%
split at 160	+13.0%	+1.2%	+7.5%	+0.6%	+2.3%	+0.0%
split at 175	+11.0%	+1.2%	+7.1%	+0.6%	+2.1%	+0.0%
split at 200	+9.4%	+1.2%	+5.7%	+0.6%	+1.9%	+0.0%

	NAS-Bench-Macro	
	full	
	time	GPU
baseline	729	3294
split at 25	+0.7%	+0.7%
split at 30	−0.9%	+0.7%
split at 35	+1.9%	+0.7%
split at 38	−0.6%	+1.0%
split at 40	−2.0%	+0.7%
split at 42	+0.4%	+0.7%
split at 45	+1.0%	+1.0%
split at 47	−3.0%	+0.3%

Table 6.1: Required training resources of the super-networks different scenarios. We list the baseline training time in seconds, and the maximum required GPU memory in MB. All variations are listed with their respective relative cost increase over the baseline. Each network was trained on a single Nvidia 1080 Ti GPU (11GB VRAM), the results are averaged over ten independent runs. The super-networks for NAS-Bench-201 and NAS-Bench-Macro have been trained for 250 and 50 epochs, respectively. The measured time is not perfectly reliable due to the random selection of candidate operations during training.

6.4.3 Ablation study

Since splitting weights comes with a reduced amount of training per weight, it is unclear whether the observed improvement window can be attributed to splitting or the reduced training. Additional NAS-Bench-201 No Zero super-networks have been trained and evaluated to answer this question. They follow the baseline schedule, except that their training has been suddenly stopped at specific epochs. As the results in Figure 6.6 show, stopping early slightly improved the architecture selection, but not as much or as systematic as the conditional weights.

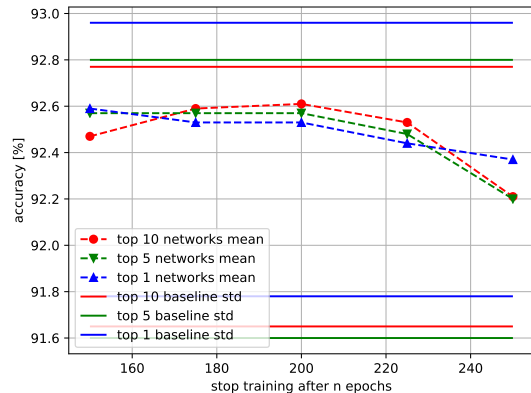


Figure 6.6: Super-networks in the NAS-Bench-201 No Zero search space. Simply stopping their training early does not produce the characteristic improvement peak of splitting weights that can be seen in Figure 6.4b. The results are also well within one standard deviation of the baseline.

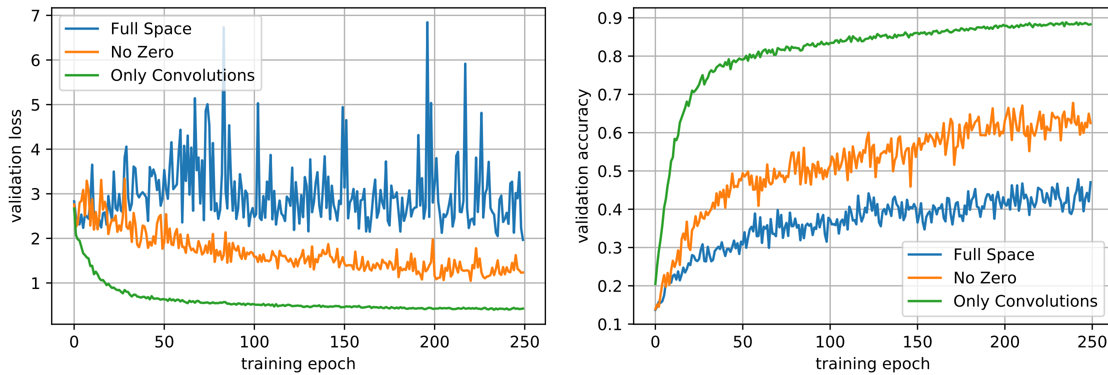


Figure 6.7: The average validation loss and accuracy during the training of the super-network baselines. While the stochastic selection of candidate operation causes spikes, the averaged network metrics keep improving.

6.5 How to find the improvement window

As seen in Figure 6.4, spitting weights with the correct timing can improve the super-network as a predictor, which results in the selection of significantly better architectures. However, this evaluation requires prior knowledge of how all selected architectures perform. If such information was available for real-world problems, there would be no need to apply architecture search.

The true difficulty of the weight splitting method is, therefore to predict the improvement window in advance. Limited to the super-network metrics during training, the clue when to split weights is likely given by validation statistics. However, as visualized in Figure 6.7, that is not necessarily the case. The improvement window, situated at around 150 epochs of training, is not matched by any eye-catching property in the validation loss or accuracy. On the contrary, all super-networks keep improving until the end of their training. This is especially interesting with respect to Figure 6.6, which shows that stopping the training early hardly affects the quality of the subsequently selected architectures. For now, the connection between the structured improvement windows and any super-network metrics is unclear.

6.6 Conclusions

In summary, this chapter described the fundamental problem of candidate co-adaptation pressure in super-networks and how conditional weights may reduce its impact. The presented weight splitting approach enables specializing the weights of every candidate operation to whichever predecessor candidate is currently selected. Even though the total number of network weights is increased multiple times, the training time and memory consumption are hardly affected.

Applying the approach to NAS-Bench-201 results in a curious phenomenon: splitting weights is most effective when done at around 60% of the total training epochs and much less so otherwise. This can be expressed as a window of opportunity when weight splitting should be applied. With the correct timing, the resulting super-networks are much better in their task of selecting high-accuracy networks. While the original baseline network selections usually achieve improvement values between 0.5 and 0.7 and are therefore somewhat far from the optimal selection possible (value of 1). the split weights can reduce this distance by about a third. On average, this results in selecting networks in the Full, No Zero, and Only Convolutions search spaces that have around 0.4%, 0.8%, and 0.15% higher accuracy, respectively.

However, an open question impairs the applicability: Finding out when to split the weights. This currently requires prior knowledge about the selected architectures that is only available in NAS benchmarks. Nonetheless, as the experiments demonstrate, conditional weights are mostly beneficial even when not split optimally.

Another topic of interest is the performance on much larger datasets. While the experiments on the MobileNet-based NAS-Bench-Macro demonstrated that improving a typical ImageNet architecture is possible, they were ultimately limited to CIFAR10. This is primarily due to the deterring costs of creating an ImageNet-based NAS benchmark, where training even a single architecture costs over 200 GPU hours.

In future experiments, finding out when to split the weights has priority. Many state-of-the-art NAS approaches are based on the Single-Path One-Shot technique but do not change their super-network. Existing approaches could be supplemented with remarkable effects if exploiting the improvement window of weight splitting becomes possible without prior knowledge.

Chapter 7

The UniNAS framework

In this final chapter before the conclusions, I will briefly introduce UniNAS (Laube, 2021), the code base used for a variety of experiments such as those in Chapter 6. The name UniNAS is a wordplay of University and Unified NAS, since the framework is extremely flexible and capable of incorporating almost any kind of architecture search approach.

7.1 Introduction

As shown in the introduction chapter of this dissertation, specifically Figure 2.1, there is an increasing supply and demand for the automatic design of neural network architectures. Consequentially, the amount of published code grows almost by the day. While code availability is obviously advantageous, this section describes technical nitpicks that can quickly dampen the enthusiasm. We start by listing the most popular NAS-related code bases and some of their common disadvantages.

7.1.1 Available frameworks

The landscape of NAS codebases is severely fragmented, owing to the vast differences between various NAS methods and the deep-learning libraries used to implement them. Some of the best supported and most widely known ones are:

- NASLib (Ruchte *et al.*, 2020), which is probably most promising framework for many research questions in the foreseeable future. It is under active development by a team from the University of Freiburg and was used for the hardware-metric prediction experiments in Chapter 5.
- Microsoft NNI (Zhang, 2019), which is the most promising framework for NAS in industry. It supports common deep-learning frameworks (TensorFlow and PyTorch, among others) and many deployment features, model compression, hyperparameter optimization, scalable search, and more. However, only a tiny handful of NAS algorithms are currently supported, limiting its usefulness in research.

- Microsoft Archai (Shah *et al.*, 2020), a more research-focused but less actively developed NAS project by Microsoft that combines several other open source code bases.
- Huawei Noah Vega (Jiajin, 2020), a multi-framework NAS pipeline primarily developed by a Chinese Huawei team. It is under active development and incorporates many NAS and hyper-parameter optimization techniques, especially for object detection tasks.
- Google TuNAS (Bender *et al.*, 2020), which demonstrated the superiority of NAS algorithms over comparably simple strategies like random search. It only features a few algorithms and no longer receives any updates, probably in favor of the unpublished PyGlove project (Peng *et al.*, 2020a).

Counterintuitively, the overwhelming majority of publicly available NAS code is not based on any such framework or service but simple and typical network training code. Such code is generally quick to implement but lacks exact comparability, scalability, and configuration power, which may be a secondary concern for many researchers. In addition, since the official code is often released late or never, and generally only in either TensorFlow (Abadi *et al.*, 2016) or PyTorch (Paszke *et al.*, 2019), popular methods are sometimes re-implemented by some third-party repositories.

Further projects include the newly available and closed-source cloud services by, e.g., Google[1] and Microsoft[2]. Since they require very little user knowledge in addition to the training data, they are excellent for deep learning in industrial environments.

7.1.2 Common disadvantages of code bases

With so many frameworks available, why start another one? The development of UniNAS started in early 2020, before most of these frameworks arrived at their current feature availability or were even made public. In addition, the frameworks rarely provide current state-of-the-art methods even now and sometimes lack the flexibility to include them easily. Further problems that UniNAS aims to solve are detailed below:

Research code is rigid As stated in Section 7.1.1, the majority of published NAS code is very simplistic. While that is an advantage to extract important method-related details, the ability to reuse the available code in another context is severely impaired. Almost all details are hard-coded, such as:

- the used gradient optimizer and learning rate schedule
- the architecture search space, including candidate operations and network topology
- the data set and its augmentations
- weight initialization and regularization techniques

[1] `https://cloud.google.com/automl/`

[2] `https://www.microsoft.com/en-us/research/project/automl/`

- the used hardware device(s) for training
- most hyper-parameters

This inflexibility is sometimes accompanied by the redundancy of several code pieces that differ slightly for different experiments or phases in NAS methods. Redundancy is a fine way to introduce subtle bugs or inconsistencies and also makes the code confusing to follow. Hard-coded details are also easy to forget, which is especially crucial in research where reproducibility depends strongly on seemingly unimportant details. Finally, if any of the hard-coded components is ever changed, such as the optimizer, configurations of previous experiments can become very misleading. Their details are generally not part of the documented configuration (since they are hard-coded), so earlier results no longer make sense.

A configuration clutter In contrast to such simplistic single-purpose code, frameworks usually offer a variety of optimizers, schedules, search spaces, and more to choose from. By configuring the related hyper-parameters, an optimizer can be easily and safely exchanged for another. Since doing so is a conscious and intended choice, it is also documented in the configuration. In contrast, the replacement of hard-coded classes was not intended when the code was initially written. The disadvantage of this approach comes with the wealth of configurable hyper-parameters, in different ways:

Firstly, the parametrization is often cluttered. While implementing more classes (such as optimizers or schedules) adds flexibility, the list of available hyper-parameters becomes increasingly bloated and opaque. The wealth of parametrization is intimidating and impractical since it is often nontrivial to understand exactly which hyper-parameters are used and which are ineffective. As an example, the widely used PyTorch Image Models framework (Wightman, 2019) (the example was chosen due to the popularity of the framework, it is no worse than others in this respect) implements an intimidating mix of regularization and data augmentation settings that are partially exclusive.[3]

Secondly, to reduce the clutter, parameters can be used by multiple mutually exclusive choices. In the case of the aforementioned PyTorch Image Models framework, one example would be the selection of gradient-descent optimizers. Sharing common parameters such as the learning rate and the momentum generally works well, but can be confusing since, once again, finding out which parameters affect which modules necessitates reading the code or documentation.

Thirdly, even with an intimidating wealth of configuration choices, not every option is covered. To simplify and reduce the clutter, many settings of lesser importance always use a sensible default value. If changing such a parameter becomes necessary, the framework configurations become more cluttered or changing the hard-coded default value again results in misleading configurations of previous experiments.

[3] `https://github.com/rwightman/pytorch-image-models/blob/ba65dfe2c6681404f35a9409f802aba2a226b761/train.py`, checked Dec. 1st 2021; see lines 177 and below.

To summarize, the hyper-parametrization design of a framework can be a delicate decision, trying for them to be complete but not cluttered. While both extremes appear to be mutually exclusive, they can be successfully united with the underlying configuration approach of UniNAS: argument trees.

Nonetheless, it is great if code is available at all. Many methods are published without any code that enables verifying their training or search results, impairing their reproducibility. Additionally, even if code is overly simplistic or accompanied by cluttered configurations, reading it is often the best way to clarify a method's exact workings and obtain detailed information about omitted hyper-parameter choices.

7.2 Argument trees

The core design philosophy of UniNAS is built on so-called *argument trees*. This concept solves the problems of Section 7.1.2 while also providing immense configuration flexibility. As its basis, we observe that any algorithm or code piece can be represented hierarchically. For example, the task to train a network requires the network itself and a training loop, which may use callbacks and logging functions.

Sections 7.2.1 and 7.2.2 briefly explain two requirements: strict modularity and a global register. As described in Section 7.2.3, this allows each module to define which other types of modules are needed. In the previous example, a training loop may use callbacks and logging functions. Sections 7.2.4 and 7.2.5 explain how a configuration file can fully detail these relationships and how the desired code class structure can be generated. Finally, Section 7.2.6 shows how a configuration file can be easily manipulated with a graphical user interface, allowing the user to create and change complex experiments without writing a single line of code.

7.2.1 Modularity

As practiced in most non-simplistic codebases, the core of the argument tree structure is strong modularity. The framework code is fragmented into different components with clearly defined purposes, such as training loops and gradient descent optimizers. Exchanging modules of the same type for one another is a simple issue, for example, the optimizers SGD, ADAM, and RMSprop. If all implemented code classes of the same type inherit from one base class (e.g., AbstractOptimizer) that guarantees specific class methods for a stable interaction, they can be treated equally. In object-oriented programming, this design is termed polymorphism.

UniNAS extends typical PyTorch (Paszke *et al.*, 2019) classes with additional functionality. An example is image classification data sets, which ordinarily do not contain information about image sizes. Adding this specification makes it possible to use fake data easily and to precompute the tensor shapes in every layer throughout the neural network.

7.2.2 A global register

A second requirement for argument trees is a global register for all modules. Its functions are:

- Allow any module to register itself with additional information about its purpose. The example code in Figure 7.1 shows this in Line 1.
- List all registered classes, including their type (task, model, optimizer, data set, and more) and their additional information (search, regression, and more).
- Filter registered classes by types and matching information.
- Given only the name of a registered module, return the class code located anywhere in the framework's files.

```
1  @Register.task(search=True)
2  class SingleSearchTask(SingleTask):
3
4    @classmethod
5    def args_to_add(cls, index=None) -> [Argument]:
6      return [
7        Argument('is_test_run', default='False', type=str),
8        Argument('seed', default=0, type=int),‘
9        Argument('save_dir', default='{path_tmp}', type=str),
10     ]
11
12   @classmethod
13   def meta_args_to_add(cls) -> [MetaArgument]:
14     methods = Register.methods.filter_match_all(search=True)
15     return [
16       MetaArgument('cls_device', Register.devices_managers, num=1),
17       MetaArgument('cls_trainer', Register.trainers, num=1),
18       MetaArgument('cls_method', methods, num=1),
19     ]
```

Figure 7.1: Example UniNAS code for a SingleSearchTask. The decorator function in Line 1 registers the class with type ”task” and additional information. The method in Line 5 returns all arguments for the task to be set in a config file. The method in Line 13 defines the local tree structure by stating how many modules of which types are needed. It is also possible to specify additional requirements, as done in Line 14.

As seen in the following sections, this functionality is indispensable to UniNAS' design. The only difficulties in building such a register is that the code should remain readable and that every module has to register itself when the framework is used. Both can be achieved by scanning through all code files whenever a new job starts, which takes less than five seconds. Python executes the decorators (see Figure 7.1, Line 1) by doing so, which handle registration in an easily readable fashion.

7.2.3 Tree-based dependency structure

A SingleSearchTask requires exactly one hardware device and exactly one training loop (named trainer, to train an over-complete super-network), which in turn may use any number of callbacks and logging mechanisms. Their relationship is visualized in Figure 7.2.

Argument trees are extremely flexible, since they allow every hierarchical one-to-any relationship imaginable. Multiple optional callbacks can be rearranged in their order and configured in detail. Moreover, module definitions can be reused in other constellations, including their requirements. The ProfilingTask does not need a training loop to measure the runtime of different network topologies on a hardware device, reducing the argument tree in size. While not implemented, a MultiSearchTask could use several trainers in parallel on several devices.

The hierarchical requirements are made available using so-called MetaArguments, as seen in Line 16 of Figure 7.1. They specify the local structure of argument trees by stating which other modules are required. To do so, writing the required module type and their amount is sufficient. As seen in Line 14, filtering the modules is also possible to allow only a specific subset. This particular example defines the upper part of the tree visualized in Figure 7.2. The names of all MetaArguments start with "cls_" which improves readability and is reflected in the visualized arguments tree (Figure 7.2, white-colored boxes).

7.2.4 Tree-based argument configurations

While it is possible to define such a dynamic structure, how can it be represented in a configuration file? Figure 7.3 presents an excerpt of the configuration that matches the tree in Figure 7.2. As stated in Lines 6 and 9 of the configuration, CudaDevicesManager and SimpleTrainer fill the roles for the requested modules of types "device" and "trainer". Lines 14 and 17 list one class of the types "logger" and "callback" each, but could provide any number of comma-separated names. Also including the stated "task" type in Line 1, the mentioned lines state strictly which code classes are used and, given the knowledge about their hierarchy, define the tree structure.

Additionally, every class has some arguments (hyper-parameters) that can be modified. SingleSearchTask defined three such arguments (Lines 7 to 9 in Figure 7.1) in the visualized example, which are represented in the configuration (Lines 2 to 4 in Figure 7.3).

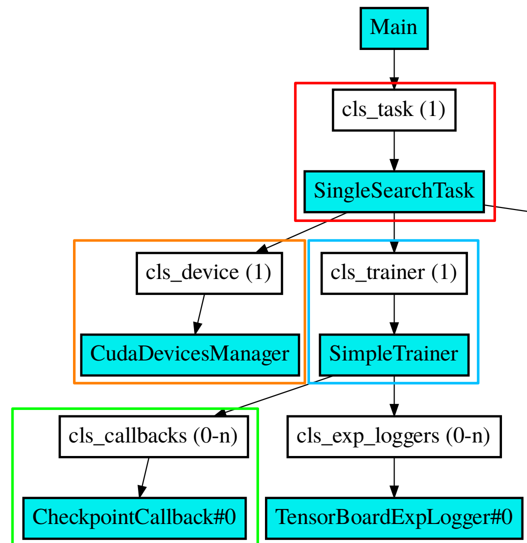


Figure 7.2: Part of a visualized SingleSearchTask configuration, which describes the training of a one-shot super-network with a specified search method (omitted for clarity, the complete tree is visualized in Figure C.3). The white colored tree nodes state the type and number of requested classes, the turquoise boxes the specific classes used. For example, the SingleSearchTask requires exactly one type of hardware device to be specified, but the SimpleTrainer accepts any number of callbacks or loggers. The colors match those in Figure 7.3.

```
1  "cls_task": "SingleSearchTask",
2  "{cls_task}.save_dir": "{path_tmp}/search/",
3  "{cls_task}.seed": 0,
4  "{cls_task}.is_test_run": true,
5
6  "cls_device": "CudaDevicesManager",
7  "{cls_device}.num_devices": 1,
8
9  "cls_trainer": "SimpleTrainer",
10 "{cls_trainer}.max_epochs": 3,
11 "{cls_trainer}.ema_decay": 0.5,
12 "{cls_trainer}.ema_device": "cpu",
13
14 "cls_exp_loggers": "TensorBoardExpLogger",
15 "{cls_exp_loggers#0}.log_graph": false,
16
17 "cls_callbacks": "CheckpointCallback",
18 "{cls_callbacks#0}.top_n": 1,
19 "{cls_callbacks#0}.key": "train/loss",
20 "{cls_callbacks#0}.minimize_key": true,
```

Figure 7.3: Example content of the configuration text-file (JSON format) for the tree in Figure 7.2. The first line in each text block specifies the used class(es), the other lines their detailed settings. For example, the SimpleTrainer is set to train for three epochs and track an exponential moving average of the network weights on the CPU.

If the configuration is missing an argument, maybe to keep it short, its default value is used. Another noteworthy mechanism in Line 2 is that ”{cls_task}.save_dir” references whichever class is currently set as ”cls_task” (Line 1), without naming it explicitly. Such wildcard references simplify automated changes to configuration files, since, independently of the used task class, overwriting ”{cls_task}.save_dir” is always an acceptable way to change the save directory. A less general but perhaps more readable notation is ”SingleSearchTask.save_dir”, which is also accepted here.

A very interesting property of such dynamic configuration files is that they contain only the hyper-parameters (arguments) of the used code classes. Adding any additional arguments will result in an error since the configuration-parsing mechanism, described in Section 7.2.5, is then unable to piece the information together. Even though UniNAS implements several different optimizer classes, any such configuration only contains the hyper-parameters of those used. Generated configuration files are always complete (contain all available arguments), sparse (contain only the available arguments), and never ambiguous.

A debatable design decision of the current configuration files, as seen in Figure 7.3, is that they do not explicitly encode any hierarchy levels. Since that information is already known from their class implementations, the flat representation was chosen primarily for readability. It is also beneficial when arguments are manipulated, either automatically or from the terminal when starting a task. The disadvantage is that the argument names for class types can only be used once (”cls_device”, ”cls_trainer”, and more); an unambiguous assignment is otherwise not possible. For example, since the SingleSearchTask already owns ”cls_device”, no other class that could be used in the same argument tree can use that particular name. While this limitation is not too significant, it can be mildly confusing at times.

Finally, how is it possible to create configuration files? Since the dynamic tree-based approach offers a wide variety of possibilities, only a tiny subset is valid. For example, providing two hardware devices violates the defined tree structure of a SingleSearchTask and results in a parsing failure. If that happens, the user is provided with details of which particular arguments are missing or unexpected. While the best way to create correct configurations is surely experience and familiarity with the code base, the same could be said about any framework. Since UniNAS knows about all registered classes, which other (possibly specified) classes they use, and all of their arguments (including defaults, types, help string, and more), an exhaustive list can be generated automatically. However, resulting in almost 1600 lines of text, this solution is not optimal either. The most convenient approach is presented in Section 7.2.6: Creating and manipulating argument trees with a graphical user interface.

Algorithm 1 Pseudo-code for building the argument tree, best understood with Figures 7.2 and 7.3 For a consistent terminology of code classes and tree nodes: If the *Task* class uses a *Trainer*, then in that context, *Trainer* the child. Lines starting with # are comments.

```
Require: Configuration                                  ▷ Content of the configuration file
Require: Register                              ▷ All modules in the code are registered

  # recursive parsing function to build a tree
  function PARSE(class, index)                            ▷ E.g. (SingleSearchTask, 0)
      node = ArgumentTreeNode(class, index)

      # first parse all arguments (hyper-parameters) of this tree node
      for (idx, argument_name) in class.get_arguments() do        ▷ E.g. (0, "save_dir")
          value = get_used_value(Configuration, class, index, argument_name)
          node.add_argument(argument_name, value)
      end for

      # then recursively parse all child classes, for each module type...
      for child_class_type in class.get_child_types() do          ▷ E.g. cls_trainer
          class_names = get_used_classes(Configuration, child_classes_type)
          Assert The number of class_names is in the specified limits

          # for each module type, check all configured classes
          for (idx, class_name) in class_names do            ▷ E.g. (0, "SimpleTrainer")
              child_class = Register.get(child_class_name)
              child_node = PARSE(child_class, idx)
              node.add_child(child_class_type, idx, child_node)
          end for
      end for
      return node
  end function

  tree = PARSE(Main, 0)          ▷ Recursively parse the tree, Main is the entry point
Ensure: every argument in the configuration has been parsed
```

7.2.5 Building the argument tree and code structure

The arguably most important function of a research code base is to run experiments. In order to do so, valid configuration files must be translated into their respective code structure. This comes with three major requirements:

- Classes in the code that implement the desired functionality. As seen in Section 7.2.3 and Figure 7.2, each class also states the types, argument names and numbers of additionally requested classes for the local tree structure.
- A configuration that describes which code classes are used and which values their parameters take. This is described in Section 7.2.4 and visualized in Figure 7.3.
- To connect the configuration content to classes in the code, it is required to reference code modules by their names. As described in Section 7.2.2 this can be achieved with a global register.

Algorithm 1 realizes the first step of this process: parsing the hierarchical code structure and their arguments from the flat configuration file. The result is a tree of *ArgumentTreeNodes*, of which each refers to exactly one class in the code, is connected to all related tree nodes, and knows all relevant hyper-parameter values. While they do not yet have actual class instances, this final step is no longer difficult.

7.2.6 Creating and manipulating argument trees with a GUI

Manually writing a configuration file can be perplexing since one must keep track of tree specifications, argument names, available classes, and more. The graphical user interface (GUI) visualized in Figures 7.4 and C.1 solves these problems to a large extent, by providing the following functionality:

- Interactively add and remove nodes in the argument tree, thus also in the configuration and class structure. Highlight violations of the tree specification.
- Setting the hyper-parameters of each node, using checkboxes (boolean), dropdown menus (choice from a selection), and text fields (other cases like strings or numbers) where appropriate.
- Functions to save and load argument trees. Since it makes sense to separate the configurations for the training procedure and the network design to swap between different constellations easily, loading partial trees is also supported. Additional functions enable visualizing, resetting, and running the current argument tree.
- A search function that highlights all matches, since the size of some argument trees can make finding specific arguments tedious.

In order to do so, the GUI manipulates *ArgumentTreeNodes* (Section 7.2.5), which can be easily converted into configuration files and code. As long as the required classes (for example, the data set) are already implemented, the GUI enables creating and changing experiments without ever touching any code or configuration files. While not among the

original intentions, this property may be especially interesting for non-programmers that want to solve their problems quickly.

Still, the current version of the GUI is a proof of concept. It favors functionality over design, written with the plain Python Tkinter GUI framework and based on little previous GUI programming experience. Nonetheless, since the GUI (frontend) and the functions manipulating the argument tree (backend) are separated, a continued development with different frontend frameworks is entirely possible. The perhaps most interesting would be a web service that runs experiments on a server, remotely configurable from any web browser.

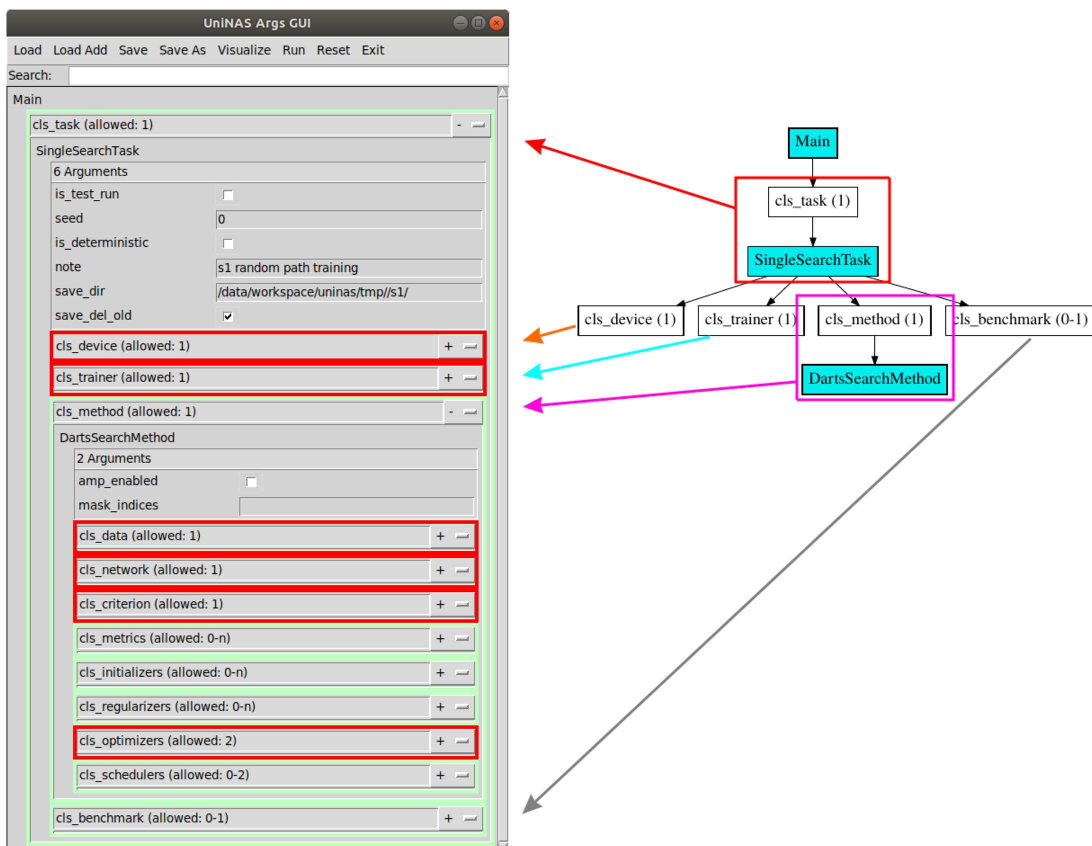


Figure 7.4: The graphical user interface (left) that can manipulate the configurations of argument trees (visualized right). Since many nodes are missing classes of some type ("cls_device", ...), their parts in the GUI are highlighted in red. The eight child nodes of DartsSearchMethod are omitted for visual clarity.

7.2.7 Using external code

There is a variety of reasons why it makes sense to include external code into a framework. Most importantly, the code either solves a standing problem or provides the users with additional options. Unlike newly written code, many popular libraries are also thoroughly optimized, reviewed, and empirically validated.

External code is also a perfect match for a framework based on argument trees. As shown in Figure 7.5, external classes of interest can be thinly wrapped to ensure compatibility, register the module, and specify all hyper-parameters for the argument tree. The integration is seamless so that finding out whether a module is locally written or external requires an inspection of its code. On the other hand, if importing the AdaBelief (Zhuang *et al.*, 2020) code fails, the module will not be registered and therefore not be available in the graphical user interface. UniNAS fails to parse configurations that require unregistered modules but informs the user which external sources can be installed to extend its functionality.

Due to this logistic simplicity, several external frameworks extend the core of UniNAS. Some of the most important ones are:

- pymoo (Blank and Deb, 2020), a library for multi-objective optimization methods.
- Scikit-learn (Pedregosa *et al.*, 2011), which implements many classical machine learning algorithms such as Support Vector Machines and Random Forests.
- PyTorch Image Models (Wightman, 2019), which provides the code for several optimizers, network models, and data augmentation methods.
- albumentations (A. Buslaev and Kalinin, 2018), a library for image augmentations.

```
1  from uninas.register import Register
2  from uninas.training.optimizers.abstract import WrappedOptimizer
3  try:
4      from adabelief_pytorch import AdaBelief
5      # if the import was successful,
6      # register the wrapped optimizer
7      @Register.optimizer()
8      class AdaBeliefOptimizer(WrappedOptimizer):
9        # wrap the original
10       ...
11
12 except ImportError as e:
13   # if the import failed,
14   # inform the user that optional libraries are not installed
15   Register.missing_import(e)
```

Figure 7.5: Excerpt of UniNAS wrapping the official AdaBelief optimizer code. The complete text has just 45 lines, half of which specify the optimizer parameters for the argument trees.

Table 7.1: The reported and reproduced results of training NAS-related network architectures on CIFAR10 (Krizhevsky *et al.*, 2009) and ImageNet (Deng *et al.*, 2009).

	CIFAR-10 error [%]		ImageNet error [%]	
Network	reported	reproduced	reported	reproduced
DARTS V1 (Liu *et al.*, 2019)	2.75	2.78	26.90	26.06
PDARTS (Chen *et al.*, 2019)	2.50	2.56		
PR-DARTS DL1 (Laube and Zell, 2019a)	2.74	2.64		
PR-DARTS DL2 (Laube and Zell, 2019a)	2.51	2.44		
ASAP (Noy *et al.*, 2020)	2.50	2.70		
MobileNet V2 (Sandler *et al.*, 2018)			28.00	27.82
Proxyless RM (Cai *et al.*, 2019)			25.40	25.34
FairNasC (Chu *et al.*, 2019a)			25.31	25.45

7.3 Reproduced results

An essential feature of any framework is to produce results. More concretely, searching and (re)training neural networks should produce equivalent results as other implementations and reported outcomes. For a variety of reasons, exact reproduction is usually not possible. As described in Section 7.1.2, incomplete information about experiments is one common cause. Different software versions or hardware setups may also have subtle effects, including the rather obvious limit to the batch sizes. Additionally, training on Nvidia GPUs is not deterministic by default to speed up the training with out-of-order processing. However, exact reproduction is generally not required. This Section briefly compares reported and reproduced results of popular methods.

Training networks Even for an architecture search code base, correctly training networks is arguably the most important. The training code parts are reused for the search routines and are also required to evaluate the performance of an architecture search result.

The reported and reproduced results of several popular NAS-related architectures are listed in Table 7.1. Unlike the originals, all reproductions use the same data, augmentations, hyper-parameters, and the SGD optimizer with cosine decay. Thus any differences in the performance are likely due to the actual architectures, not their respective training setups. Nonetheless, the reproduced results are generally close to the respective originals.

Searching for networks Unfortunately, evaluating the correctness of NAS methods is much more complicated. They are often dependent on delicate hyper-parameter con-

figurations and other non-obvious factors. In terms of the primarily used Single-Path One-Shot (SPOS) based approaches, figuring out whether a problem is due to the super-network or the hyper-parameter optimization method using it is an additional difficulty. In fact, reproducing the reported SPOS results is known to be difficult[4]. Consequentially, most comparisons are limited to benchmark problems (see Chapter 2.5.1).

Dong and Yang (2020) present NAS-Bench-201 search results for several different algorithms and data sets in Table 5 of their paper. Exact reproductions of the benchmark results were not attempted, especially since SPOS is not among the evaluated methods. Nonetheless, our own SPOS-based results (Chapter 6, Figure 6.4a, 92.2%) perform considerably better than the baselines except for GDAS (Dong and Yang, 2019) and the methods with access to ground-truth test results.

7.4 Dynamic network designs

As seen in the previous sections, the unique design of UniNAS enables powerful customization of all components. In most cases, a significant portion of the architecture search configuration belongs to the network design. The FairNAS search example in Figure C.3 contains 25 configured classes, of which 11 belong to the search network. While it would be easy to create a single configurable class for each network architecture of interest, that would ignore the advantages of argument trees. On the other hand, there are many technical difficulties with highly dynamic network topologies. Some of them are detailed below.

7.4.1 Decoupling components

In many published research codebases, network and architecture weights jointly exist in the network class. This design decision is disadvantageous for multiple reasons. Most importantly, changing the network or NAS method requires a lot of manual work. The reason is that different NAS methods need different amounts of architecture parameters, use them differently, and optimize them in different ways. For example:

- DARTS (Liu *et al.*, 2019) requires one weight vector per architecture choice. They weigh all different paths, candidate operations, in a sum. Updating the weights is done with an additional optimizer (ADAM), using gradient descent.
- MDENAS (Zheng *et al.*, 2019) uses a similar vector for a weighted sample of a single candidate operation that is used in this particular forward pass. Global network performance feedback is used to increase or decrease the local weightings.

[4]`https://nni.readthedocs.io/en/v2.5/NAS/SPOS.html`, see the current reproduction results at the bottom of the website.

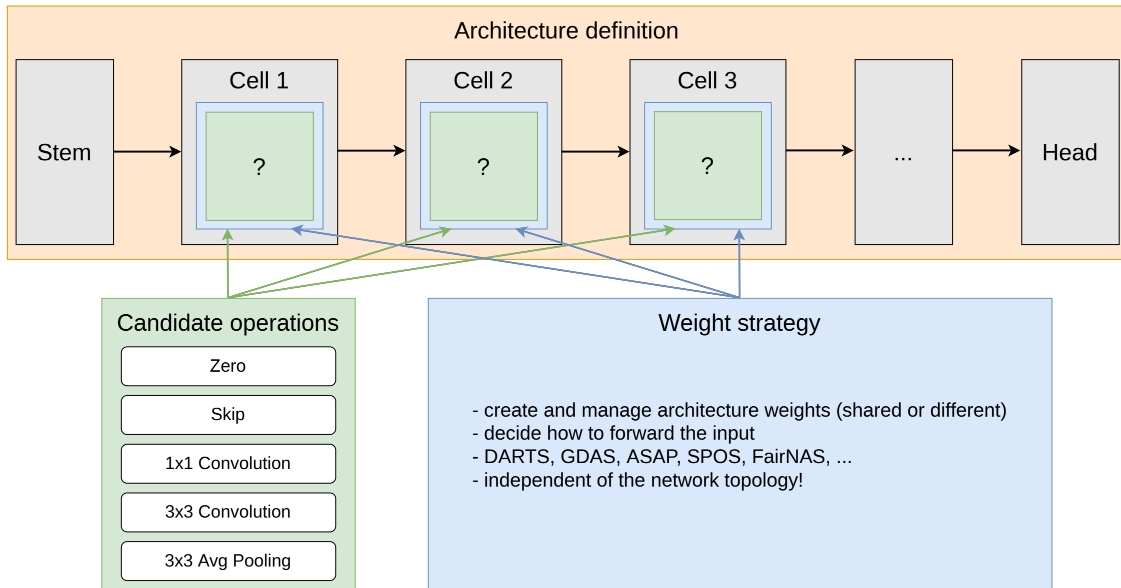


Figure 7.6: The network and architecture weights are decoupled.
Top: The structure of a fully sequential super-network. Every layer (cell) uses the same set of candidate operations and weight strategy. **Bottom left**: One set of candidate operations that is used multiple times in the network. This particular experiment uses the NAS-Bench-201 candidate operations. **Bottom right**: A weight strategy that manages everything related to the used NAS method, such as creating the architecture weights or which candidates are used in each forward pass.

- Single-Path One-Shot (Guo *et al.*, 2020) does not use weights at all. Paths are always sampled uniformly randomly. The trained network is used as an accuracy prediction model and used by a hyper-parameter optimization method.
- FairNAS (Chu *et al.*, 2019a) extends Single-Path One-Shot to make sure that all candidate operations are used frequently and equally often. It thus needs to track which paths are currently available.

The same is also true for the set of candidate operations, which affect the sizes of the architecture weights. Once the definitions of the search space, the candidate operations, and the NAS method (including the architecture weights) are mixed, changing any part is tedious. Therefore, strictly separating them is the best long-term approach. Similar to other frameworks presented in Section 7.1.1, architectures defined in UniNAS do not use an explicit set of candidate architectures but allow a dynamic configuration. This is supported by a *WeightStrategy* interface, which handles all NAS-related operations such as creating and updating the architecture weights. The interaction between the architecture definition, the candidate operations, and the weight strategy is visualized in Figure 7.6.

The easy exchange of any component is not the only advantage of this design. Some NAS methods, such as DARTS, update network and architecture weights using different gradient descent optimizers. Correctly disentangling the weights is trivial if they are already organized in decoupled structures but hard otherwise. Another advantage is that standardizing functions to create and manage architecture weights makes it easy to present relevant information to the user, such as how many architecture weights exist, their sizes, and which are shared across different network cells. An example is presented in Figure C.2.

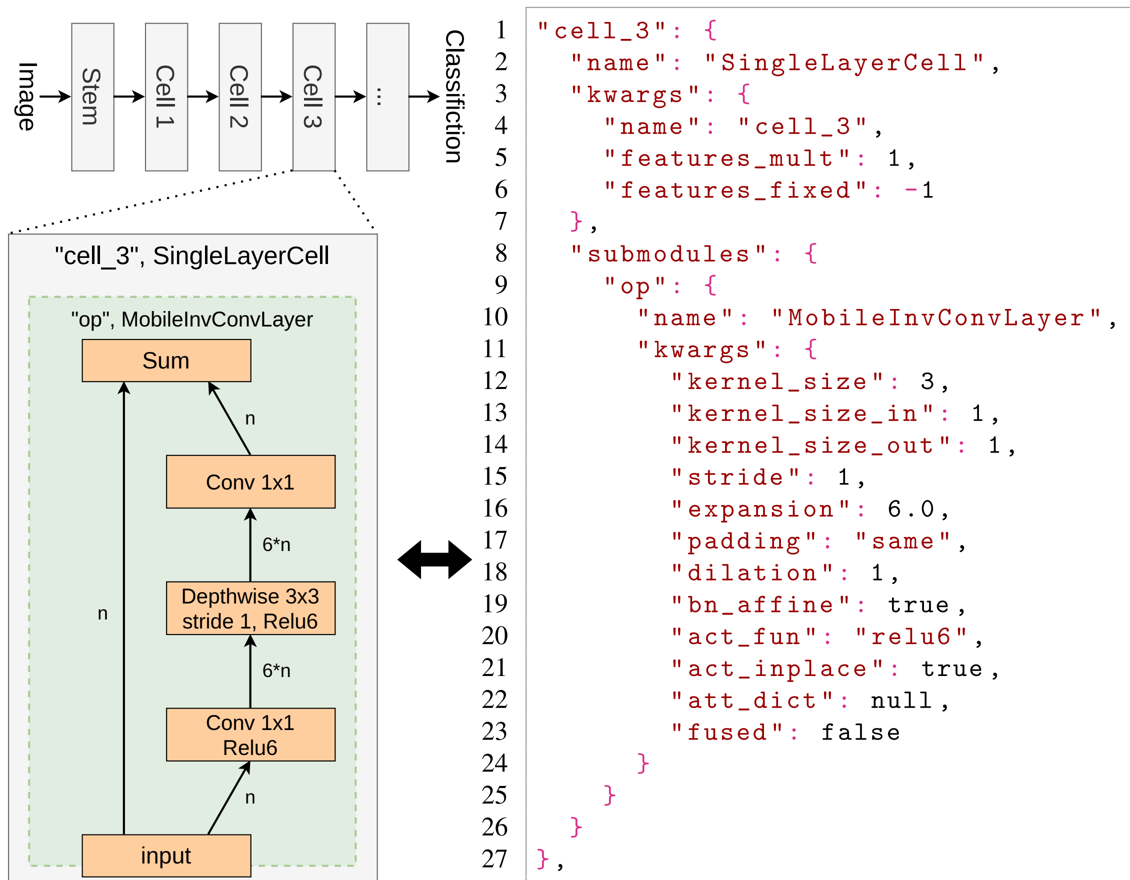


Figure 7.7: A high-level view on the MobileNet V2 architecture (Sandler *et al.*, 2018) in the top left, and a schematic of the inverted bottleneck block in the bottom left. This design uses two 1×1 convolutions to change the channel count n by an expansion factor of 6, and a spatial 3×3 convolution in their middle. The text on the right-hand side represents the cell structure by referencing the modules by their names ("name") and their keyworded arguments ("kwargs").

7.4.2 Saving, loading, and finalizing networks

As mentioned before, argument trees enable a detailed configuration of every aspect of an experiment, including the network topology itself. As visualized in Figure C.3, such network definitions can become almost arbitrarily complex. This becomes disadvantageous once models have to be saved or loaded or when super-networks are finalized into discrete architectures. Unlike TensorFlow (Abadi *et al.*, 2016), the used PyTorch (Paszke *et al.*, 2019) library saves only the network weights without execution graphs. External projects like ONNX (Bai *et al.*, 2019) can be used to export limited graph information but not to rebuild networks using the same code classes and context.

The implemented solution is inspired by the official code[5] of ProxylessNAS (Cai *et al.*, 2019), where every code module defines two functions that enable exporting and importing the entire module state and context. As typical for hierarchical structures, the state of an outer module contains the states of all modules within. An example is visualized in Figure 7.7, where one cell in the famous MobileNet V2 architecture is represented as readable text. The global register can provide any class definition by name (see Section 7.2.2) so that an identical class structure can be created and parameterized accordingly.

The same approach that enables saving and loading arbitrary class compositions can also be used to change their structure. More specifically, an over-complete super-network containing all possible candidate operations can export only a specific configuration subset. The network recreated from this reduced configuration is the result of the architecture search. This is made possible since the weight strategy controls the use of all candidate operations, as visualized in Figure 7.6. Similarly, when their configuration is exported, the weight strategy controls which candidates should be part of the finalized network architecture. In another use case, some modules behave differently in super-networks and finalized architectures. For example, Linear Transformers (Chu *et al.*, 2019b) supplement skip connections with linear 1×1 convolutions in super-networks to stabilize the training with variable network depths. When the network topology is finalized, it suffices to simply export the configuration of a skip connection instead of their own.

Another practical way of rebuilding code structures is available through the argument tree configuration, which defines every detail of an experiment (see Section 7.2.4). Parsing the network design and loading the trained weights of a previous experiment requires no further user interaction than specifying its save directory. This specific way of recreating experiment environments is used extensively in *Single-Path One-Shot* tasks. In the first step, a super-network is trained to completion. Afterward, when the super-network is used to make predictions for a hyper-parameter optimization method (such as Bayesian optimization or evolutionary algorithms), the entire environment of its training can be recreated. This includes the network design and the dataset, data augmentations, which parts were reserved for validation, regularization techniques, and more.

[5] `https://github.com/mit-han-lab/proxylessnas/tree/master/proxyless_nas`

7.5 Discussion and Conclusions

This chapter presented the underlying concepts of UniNAS, a PyTorch-based framework with the ambitious goal of unifying a variety of NAS algorithms in one codebase. Even though the use cases for this framework changed over time, mostly from DARTS-based to SPOS-based experiments, its underlying design approach made reusing old code possible at every step. However, several technical details could be changed or improved in hindsight. Most importantly, configuration files should reflect the hierarchy levels (see Section 7.2.4) for code simplicity and to avoid concerns about using module types multiple times. The current design favors readability, which is now a minor concern thanks to the graphical user interface. Other considered changes would improve the code readability but were not implemented due to a lack of necessity and time.

In summary, the design of UniNAS fulfills all original requirements. Modules can be arranged and combined in almost arbitrary constellations, giving the user an extremely flexible tool to design experiments. Furthermore, using the graphical user interface does not require writing even a single line of code. The resulting configuration files contain only the relevant information and do not suffer from a framework with many options. These features also enable an almost arbitrary network design, combined with any NAS optimization method and any set of candidate operations. Despite that, networks can still be saved, loaded, and changed in various ways. Although not covered in this chapter, several unit tests ensure that the essential framework components keep working as intended.

While it is unlikely that UniNAS will be developed further, primarily due to a lack of development time, other projects will probably adopt argument trees. They are an elegant way to avoid suffering from feature bloat while also making it possible to visualize and manipulate experiments without any coding knowledge. This goal aligns well with AutoML, which is also intended to make machine learning available to a broader audience.

Chapter 8

Conclusions

8.1 Summary

The topic of this dissertation is the study and improvement of Neural Architecture Search, the automated design of neural networks. Reinvented less than ten years ago, this field of study changes rapidly and largely lacks established practices and traditions. The chapters of this dissertation reflect some of these changes by considering different search space designs, methods that operate on them, and strategies to compare them.

Based on the observation that the initial NASNet networks are complicated and slow, we used the ShuffleNet V2 guidelines for efficient network designs to reinvent the original search space in Chapter 3. Compared to their ENAS baseline, the discovered ShuffleNASNets models achieve the same accuracy while being significantly less complex, twice as fast, and using fewer network parameters.

In the ideal scenario, however, NAS can find suitable architectures without creating an elaborate search space in the first place. Morphism-based approaches grow and mutate networks from scratch but generally perform worse than methods using fixed super-networks. We combined many of their advantages in Chapter 4, where network morphisms are periodically used to improve an otherwise fixed super-network-based search space. The proposed *Prune and Replace* approach considers many more candidate operations than otherwise possible, by gradually adjusting properties of Convolutions such as kernel sizes and strides. Although using a more open and challenging search space than the DARTS baseline, our PR-DARTS method discovered higher accuracy models in a shorter time.

Chapters 3 and 4 considered the number of model parameters and FLOPs as important metrics but did not explicitly optimize them. In contrast, many modern approaches consider the multi-objective problem of maximizing accuracy while minimizing latency or energy consumption. Even though prediction models are widely used to estimate such device- and architecture-specific metrics, a comprehensive study that compares predictors was missing. We present such a study in Chapter 5 and find that, in most cases, the extensively used Lookup Tables are easily outperformed even by Linear Regression. Across a wide range of training set sizes, MLPs perform best. We also extensively simulated a multi-objective selection of architectures from a search space, using predictors of

different quality, data sets, and numbers of considered architectures. This allowed us to quantify how different prediction models affect the NAS results and demonstrate the benefit of considering more candidate architectures, even at the cost of using lower-quality predictors.

An improvement approach of a specific predictor, the super-network, is evaluated in Chapter 6. We observed that the widely used single-path methods implicitly pressure all candidate operations in a layer to co-adapt, which limits their capacity. We proposed conditional architecture weights to approach this issue since they allow candidate operations to specialize towards each other, in different constellations and across network layers. This can be achieved with only a minor effect on the training time and required memory by a process of *weight splitting*. If done correctly, the trained super-networks have a higher ranking correlation and select better top-performing models. However, since it is not yet clear how to predict this moment in advance, an application in real-world problems is currently tricky.

As exemplary seen from the presented works, there is a great variety in NAS search spaces, algorithm designs, components, their underlying principles, and more. The variety results in an almost equally high fragmentation of published code, despite the availability of multiple well-supported NAS frameworks. We present UniNAS in Chapter 7, a framework designed to handle almost arbitrarily complex and diverse algorithm designs. This is made possible with argument trees, a concept that allows code modules to be reused and combined in various ways. We also demonstrated that argument trees can be manipulated with a graphical user interface to create and change complex experiments without writing a single line of code.

8.2 Discussions and Future Work

Even though Neural Architecture Search ideally solves the problem at hand without requiring search space adaptions or any domain knowledge, that is currently not quite possible. Most fundamental design decisions are left to a human engineer, such as the initial network structure, layers, channels, and available candidate operations; although some NAS methods can change these in limited ways. Like classic hyper-parameter optimization methods that find the best learning rate in a given interval, NAS methods find promising architectures in a defined search space. A suitable design, as systematically developed in Chapters 3, is therefore indispensable.

A related topic of equally high importance is device awareness. Depending on the hardware platform, network topologies and candidate operations differ significantly by their execution times, energy consumption, and more (Benmeziane *et al.*, 2021). Multi-objective NAS methods can find suitable task- and device-specific architectures, which makes them a promising approach for many current and future applications. Vital components of such multi-objective methods are prediction models, likewise for hardware metrics and network loss. The study presented in Chapter 5 provides comprehensive base-

lines and recommendations, which will serve as a guideline for future methods. Meta-learning and layer-wise prediction approaches were not included but are an interesting research direction for proceeding experiments. We also investigated an improvement approach for an essential accuracy prediction model in Chapter 6, the super-networks. The proposed *conditional super-network weights* show promising results, despite their current limitations.

Even though carefully and systematically engineered search spaces and NAS methods are dominant right now, that may change shortly. Freely grown and mutated networks do not reach the same performance as their competition but require far less domain knowledge and general understanding of machine learning. Therefore, they are an ideal starting point and baseline for many uncommon real-world problems that an established architecture can not easily solve. Furthermore, if their performance is acceptable, investing in a sophisticated search space is no longer necessary. Although not quite usable yet, Real *et al.* (2020) showed that even entire machine learning algorithms could be automatically created and improved from scratch, not just their models. We presented an initial work for another potential future approach in Chapter 4, a hybrid of fixed and weakly defined search spaces. If methods such as the proposed *prune and replace* approach can efficiently modify the number of layers and their connections, not just the candidate operations, manually designing search spaces may no longer be needed.

Appendix A

What to expect of hardware metric predictors in NAS

A.1 Encodings and Predictors

Data encodings

Every architecture $a \in \mathcal{A}$ requires a unique representation, which depends on the used predictor. The common encoding types are:

Adjacency one-hot: Each architecture a is uniquely defined by the chosen candidate operation on every path. For example, each architecture in NAS-BENCH-201 consists of a repeated cell structure, which has five candidate operations on each of the six paths. Therefore there are $5^6 = 15625$ unique architectures, each referenced by a sequence of operation-indices such as [0 1 2 3 4 0]. Many predictors perform better if the sequence is presented as a one-hot encoding, which is in this case [10000 01000 00100 00010 00001 10000].

Similarly, the **path-encoding** (used by BANANAS) is a one-hot representation over the used candidate operation on all possible paths between any two network graph nodes. Since the connectivity within cells for HW-NAS-Bench and TransNAS-Bench-101 is fixed, it provides no more information than the adjacency one-hot encoding (all other paths would use the Zero operation). If the connectivity can be adjusted more freely, as in the NAS-Bench-101 search space, the additional information may improve the fit.

The encodings for **BONAS**, **GCN**, and **NAO** each provide further information in addition to the Adjacency one-hot vector, most notably the adjacency-matrix. This $\{0, 1\}^{(N+2)\times(N+2)}$ matrix lists describes which of the N architecture paths (rows) serves as inputs for each other path (column), and also includes input/output.

Predictors

We briefly describe the 18 predictor methods in our experiments. We adopt their implementations from the NASLib library, which we extend with Linear Regression, Ridge Regression, and Support Vector Machines from the scikit-learn package; and a simple Lookup Table implementation. Unless specified otherwise, the methods use the adjacency one-hot encoding.

- **BANANAS** An ensemble of three MLP models with five to 20 layers, each using the path-encoding (White *et al.*, 2019).
- **Bayesian Linear Regression** A bayesian model that assumes (1) a linear dependency between inputs and outputs, and (2) that the samples are normally distributed (Bishop, 2007).
- **BOHAMIANN** A bayesian inference predictor using stochastic gradient Hamiltonian Monte Carlo (SGHMC) to sample from a bayesian neural network (Springenberg *et al.*, 2016).
- **BONAS** Bayesian Optimization for NAS (Shi *et al.*, 2020) uses a GCN predictor within an outer loop of bayesian optimization, as a meta-learning task. The GCN requires encoding the adjacency matrix of each architecture.
- **Gaussian Process** A simple model that assumes a joint Gaussian distribution underlying the training data (Rasmussen, 2003).
- **GCN** A Graph Convolutional Network that makes use of an adjacency-matrix encoding of each architecture (Wen *et al.*, 2020).
- **Linear Regression** A simple model that assumes an independent value/cost for each operation/layer, which only need to be summed up. Unlike the Lookup Table model, it uses a least-square fit on the training data.
- **Lookup Table** The most simple and perhaps widely used model for differentiable architecture selection. It generally assumes a single baseline architecture (e.g. [001 001] in adjacency one-hot encoding), and a lookup matrix $\mathbb{R}^{(\text{num layers})\times(\text{num candidates})}$ that contains the increases/reductions in the metric for each layer and candidate operation. The metric value of a new architecture can be predicted with a simple sum over the respective matrix entries and the baseline value. The model is obtained from measuring either each candidate operation in isolation, or by computing the differences between the baseline architecture and specific variations (e.g. [010 001] or [100 001], to measure the first candidates). This model always requires $1+(\textit{num layers})\cdot(\textit{num candidates}-1)$ neighbored architectures to fit. We detail the resulting correlation values for each used dataset in Appendix A.3.

- **LGBoost** Light Gradient Boosting Machine (LightGBM or LGBoost, Ke *et al.* (2017)) is a lightweight gradient-boosted decision tree model.
- **MLP** We use fully-connected Multi Layer Perceptrons in two size-categories.
- **NAO** NAO (Luo *et al.*, 2018) uses an encoder-decoder topology, which encodes/-compresses an architecture to a continuous representation, and decodes it again. This representation is further used to make architecture predictions.
- **NGBoost** Natural Gradient Boosting (NGBoost, Duan *et al.* (2020)) is a gradient-boosted decision tree model that uses natural gradients to estimate uncertainty.
- **Ridge Regression** Ridge Regression (Saunders *et al.*, 1998) extends the Linear Regression least-squares fit with a regularization term that serves as bias-variance tradeoff.
- **Random Forests** An ensemble of decision trees (Liaw *et al.*, 2002).
- **Sparse Gaussian Process** an approximation of Gaussian Processes that summarizes training data (Candela and Rasmussen, 2005).
- **Support Vector Machine** A model that maps its inputs to a high-dimensional space, where training samples are used as support-vectors for decision-boundaries (Cortes and Vapnik, 1995).
- **XGBoost** eXtreme Gradient Boosting (XGBoost, Chen and Guestrin (2016)) is a gradient-boosted decision tree model.

A.2 Hyperparameters

We list our default and hyper-parameter sample ranges in Table A.1. For comparability with White *et al.* (2021), we only change the values of newly introduced parameterized predictors: Ridge Regression, Support Vector Machines, and small MLPs.

Table A.1: Hyper-parameter ranges and default values of the configurable predictors

Model	Hyper-parameter	Range/Choice	Log-transform	Default
BANANAS	Num. Layers	$[5, 25]$	false	20
	Layer width	$[5, 25]$	false	20
	Learning rate	$[0.0001, 0.1]$	true	0.001
BONAS	Num. Layers	$[16, 128]$	true	64
	Batch size	$[32, 256]$	true	128
	Learning rate	$[0.00001, 0.1]$	true	0.0001
GCN	Num. Layers	$[64, 200]$	true	144
	Batch size	$[5, 32]$	true	7
	Learning rate	$[0.00001, 0.1]$	true	0.0001
	Weight decay	$[0.00001, 0.1]$	true	0.0003
LGBoost	Num. leaves	$[10, 100]$	false	31
	Learning rate	$[0.001, 0.1]$	true	0.05
	Feature fraction	$[0.1, 1]$	false	0.9
MLP (small)	Num. layers	$[2, 5]$	false	3
	Layer width	$[16, 128]$	true	32
	Learning rate	$[0.0001, 0.1]$	true	0.001
	Activation function	{relu, tanh, hardswish}		relu
MLP (huge)	Num. layers	$[5, 25]$	false	20
	Layer width	$[5, 25]$	false	20
	Learning rate	$[0.0001, 0.1]$	true	0.001
NAO	Num. layers	$[16, 128]$	true	64
	Batch size	$[32, 256]$	true	100
	Learning rate	$[0.00001, 0.1]$	true	0.001
NGBoost	Num. estimators	$[128, 512]$	true	64
	Learning rate	$[0.001, 0.1]$	true	0.081
	Max depth	$[1, 25]$	false	6
	Max features	$[0.1, 1]$	false	0.79
Ridge Regression	Regularization α	$[0.25, 2.5]$	false	1.0
Random Forests	Num. estimators	$[16, 128]$	true	116
	Max features	$[0.1, 0.9]$	true	0.17
	Min samples (leaf)	$[1, 20]$	false	2
	Min samples (split)	$[2, 20]$	true	2
Support Vector Machine	Regularization C	$[0.5, 1.5]$	false	1.0
	Kernel	{linear, poly, rbf, sigmoid}		rbf
XGBoost	Max depth	$[1, 15]$	false	6
	Min child weight	$[1, 10]$	false	1
	Col sample (tree)	$[0, 1]$	false	1
	Learning rate	$[0.001, 0.5]$	true	0.3
	Col sample (level)	$[0, 1]$	false	1

A.3 Selection of datasets

	Linear Regression										XGBoost	LUT
	11	25	55	124	276	614	1366	3036	6748	15000	15000	-
ImageNet16-120-raspi4_latency	0.324	0.205	0.606	0.676	0.705	0.716	0.715	0.723	0.728	0.729	0.757	0.443
cifar100-pixel3_latency	0.392	0.292	0.732	0.780	0.797	0.803	0.806	0.809	0.812	0.812	0.877	0.484
cifar10-edgegpu_latency	0.370	0.258	0.724	0.790	0.806	0.819	0.820	0.822	0.830	0.829	0.926	0.175
cifar100-edgegpu_energy	0.376	0.275	0.732	0.793	0.812	0.821	0.821	0.823	0.831	0.831	0.920	0.221
ImageNet16-120-eyeriss arith. int.	0.369	0.293	0.748	0.805	0.817	0.827	0.825	0.832	0.843	0.846	0.970	0.861
cifar10-pixel3_latency	0.388	0.300	0.733	0.780	0.797	0.805	0.805	0.810	0.813	0.813	0.878	0.475
cifar10-raspi4_latency	0.393	0.315	0.740	0.787	0.799	0.805	0.807	0.810	0.813	0.813	0.890	0.462
cifar100-raspi4_latency	0.393	0.308	0.744	0.786	0.801	0.807	0.810	0.810	0.814	0.814	0.888	0.445
ImageNet16-120-pixel3_latency	0.398	0.312	0.739	0.786	0.799	0.807	0.809	0.812	0.815	0.816	0.884	0.509
cifar100-edgegpu_latency	0.375	0.268	0.728	0.793	0.810	0.821	0.820	0.822	0.831	0.831	0.924	0.191
cifar10-edgegpu_energy	0.375	0.284	0.728	0.792	0.810	0.821	0.823	0.824	0.831	0.831	0.922	0.183
ImageNet16-120-edgegpu_energy	0.377	0.281	0.733	0.797	0.814	0.825	0.825	0.826	0.834	0.833	0.926	0.280
ImageNet16-120-edgegpu_latency	0.379	0.264	0.737	0.799	0.817	0.826	0.826	0.828	0.836	0.835	0.938	0.277
cifar10-eyeriss arith. int.	0.384	0.296	0.757	0.811	0.826	0.835	0.832	0.843	0.854	0.854	0.969	0.826
cifar100-eyeriss arith. int.	0.384	0.297	0.757	0.811	0.826	0.835	0.833	0.844	0.855	0.856	0.971	0.830
ImageNet16-120-fpga_latency	0.443	0.494	0.904	0.936	0.947	0.951	0.948	0.951	0.952	0.952	0.983	0.965
ImageNet16-120-fpga_energy	0.443	0.494	0.905	0.935	0.947	0.951	0.948	0.951	0.952	0.952	0.983	0.965
ImageNet16-120-eyeriss_latency	0.457	0.937	0.953	0.954	0.954	0.954	0.953	0.953	0.954	0.954	0.952	0.989
cifar10-eyeriss_latency	0.461	0.943	0.959	0.959	0.960	0.960	0.959	0.960	0.960	0.960	0.958	0.995
cifar100-eyeriss_latency	0.462	0.946	0.963	0.963	0.963	0.963	0.963	0.963	0.964	0.963	0.962	0.998
cifar10-eyeriss_energy	0.456	0.967	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.985	0.975	0.996
ImageNet16-120-eyeriss_energy	0.458	0.967	0.985	0.985	0.986	0.985	0.986	0.985	0.985	0.986	0.972	0.998
cifar100-eyeriss_energy	0.457	0.967	0.985	0.985	0.985	0.986	0.985	0.986	0.986	0.986	0.976	0.998
cifar10-fpga_energy	0.458	0.973	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.986	0.999
cifar100-fpga_energy	0.458	0.973	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.986	0.999
cifar100-fpga_latency	0.457	0.973	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.986	0.999
cifar10-fpga_latency	0.457	0.973	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.987	0.986	0.999

Table A.2: Kendall's Tau test correlation for Linear Regression, XGBoost, and Lookup Tables (LUT) on all HW-NAS-Bench datasets (rows), for different amounts of available training data (columns), tested on the remaining 625 samples. The Lookup Table model is tested on all 15625 architectures. We selected the five data sets at the top.

HW-NAS-Bench: To select five datasets that are (1) non-linear and (2) different from one another, we first fit Linear Regression to every available dataset, with the results listed in Table A.2. The bottom 12 datasets can be accurately fit with only 25 training samples, so they are not very interesting as a challenge. On these datasets, the Lookup Table model achieves exceptional performance. Since the networks for CIFAR10, CIFAR100 and ImageNet16-120 only differ slightly, their measurements on the same device and metric (e.g. raspi4 latency) is very similar. To improve the generalizability of our results, we thus select datasets on different devices and metrics, which are listed at the top of Table A.2. As displayed in Figure A.1, their data distributions are generally different.

TransNAS-Bench-101: Since the latency measurements of the architectures is generally very similarly distributed (see Figure A.2), it is not necessary to train the predictors on all of them. We select all data sets that provide the *test_loss* and *inference_time* attributes for all architectures, resulting in exactly the five datasets listed in Section 5.5 (the other two datasets contain more specific test losses).

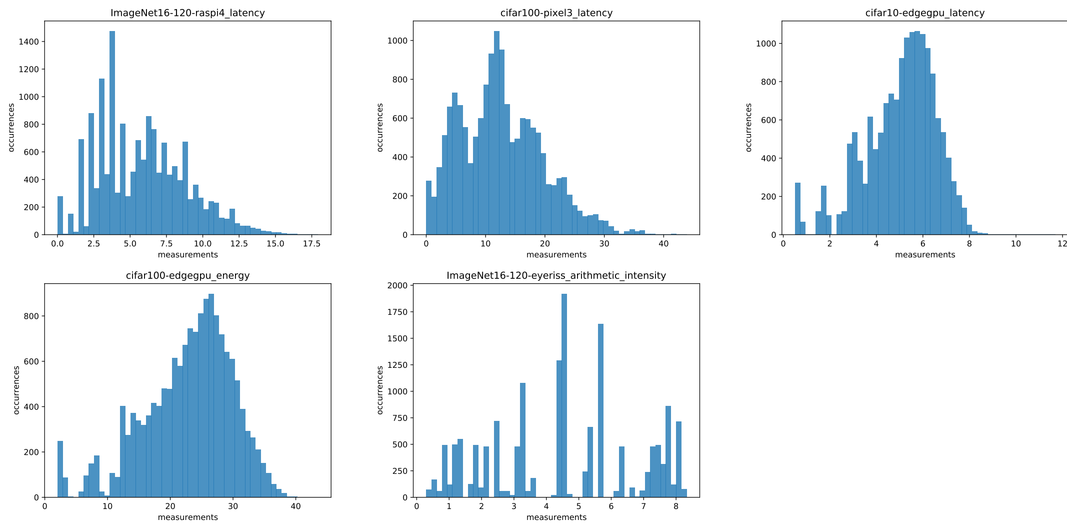


Figure A.1: How the data of each selected HW-NAS-Bench dataset is distributed (not normalized).

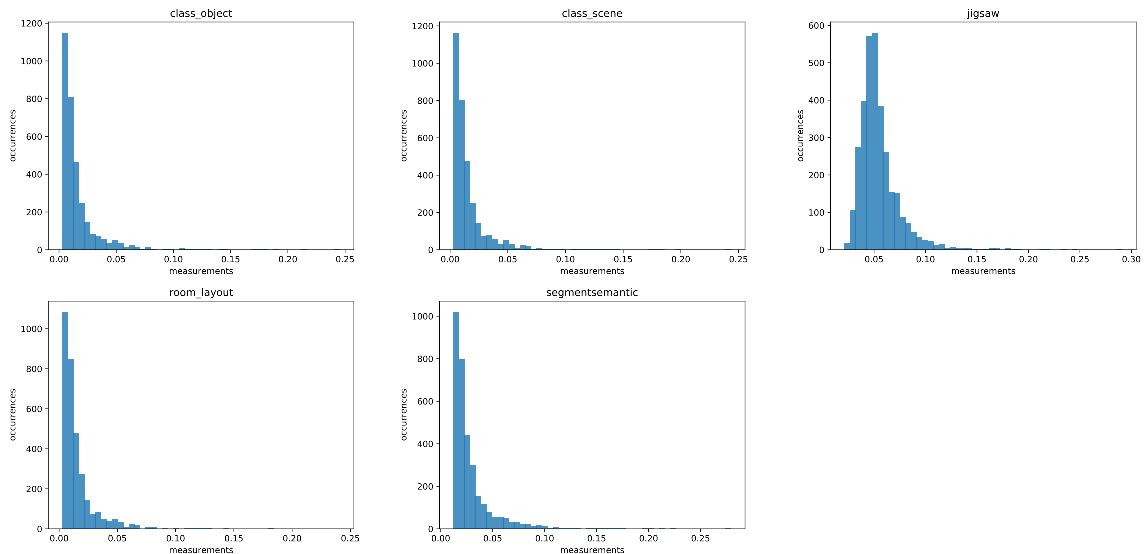


Figure A.2: How the data of each selected TransNAS-Bench-101 dataset is distributed (not normalized). Since all architectures are measured for latency on the same hardware, the resulting datasets are much less diverse than the HW-NAS-Bench ones.

	Linear Regression										XGBoost	LUT
	9	18	34	65	123	234	442	837	1585	2999	2999	-
jigsaw	0.201	0.227	0.410	0.535	0.586	0.605	0.616	0.624	0.631	0.632	0.661	0.201
class_object	0.268	0.262	0.518	0.646	0.711	0.741	0.759	0.771	0.780	0.780	0.828	0.701
room_layout	0.275	0.271	0.527	0.653	0.721	0.753	0.768	0.780	0.789	0.789	0.896	0.685
class_scene	0.275	0.268	0.527	0.653	0.721	0.755	0.768	0.782	0.789	0.790	0.907	0.710
segmentsemantic	0.282	0.259	0.545	0.684	0.746	0.780	0.798	0.809	0.816	0.818	0.871	0.726

Table A.3: Kendall's Tau test correlation for Linear Regression and XGBoost on the five used TransNAS datasets (rows), for different amounts of available training data (columns), tested on the remaining 256 samples. The Lookup Table model (LUT) is tested on all 3256 architectures.

A.4 Predictor fit time

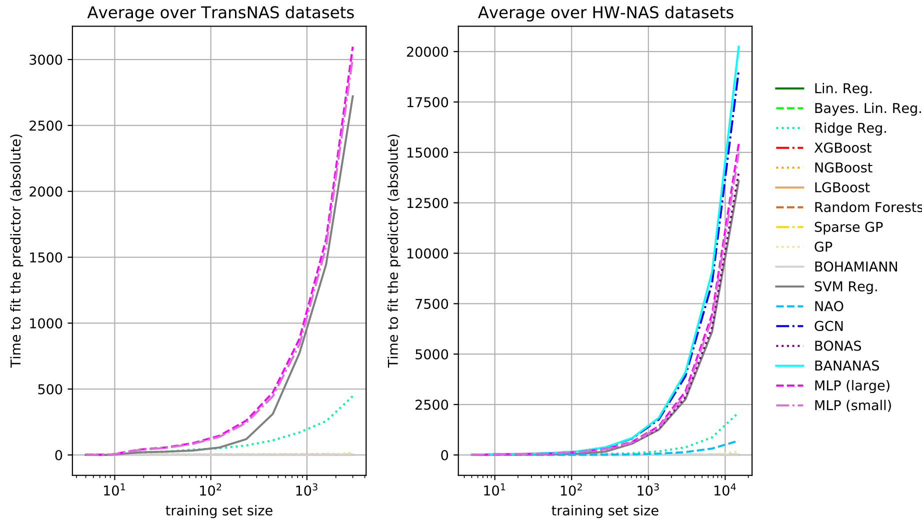


Figure A.3: Fit time (in seconds) of predictors to data, depending on the training set size. By far the most expensive methods are network-based. However, a significant portion of this time is spent on the hyper-parameter optimization prior to the actual fitting.

A.5 Approximating predictor mistakes

Intuitively, the predictor deviation distributions (see Figures 5.4 and A.4) generally resemble a normal distribution. However, most predictors:

(1) Have a notable peak, sometimes off-center (e.g. at x=0.2)
(2) Have less density than a normal distribution almost everywhere else
(3) Have some outliers (e.g. at x>1.5) that are extremely unlikely for a normal distribution

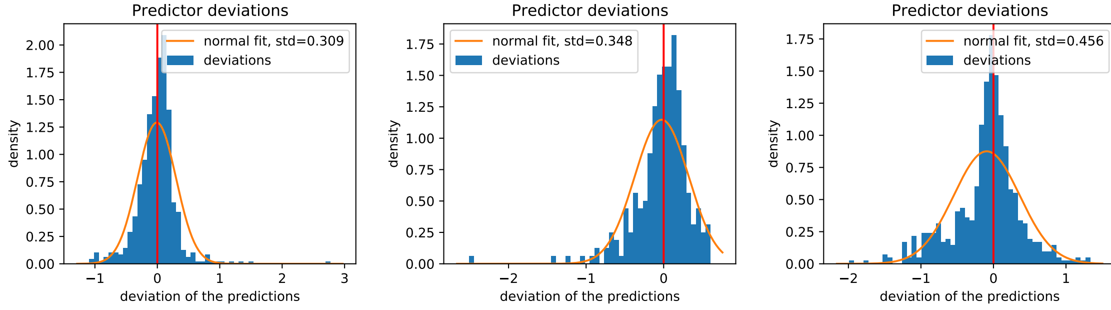


Figure A.4: Further examples of predictor deviation distributions, as visualized in the center of Figure 5.4. **Left:** Linear Regression on CIFAR100, edgegpu, energy consumption. **Center:** Support Vector Machine on Jigsaw. **Right:** small MLP on ImageNet16-120, raspi4, latency.

Table A.4: P-values of different distributions, trying to fit the distribution of all predictor mistakes according to a t-test. Larger values are better, but comparing many empirically sampled points with a true density function tends to push the p-values to 0.

	p-value
normal	0.028
cauchy	0.030
lognorm	0.028
t	0.028
uniform	0.037

We measured the p-value for different distributions on the first 100 test samples using a T-Test every time we evaluated a predictor. The average statistics can be found in Table A.4. Since many empirical observations generally push the p-value to 0, this only serves to compare them to each other. We find that the outliers (3) appear often enough and are so unlikely to happen for a normal distribution that even a uniform distribution has higher statistical support. Consequentially, we approximate the common predictor deviations by sampling from a mixed distribution that addresses (1) to (3).

This mixed distribution consists of two Normal distributions (N_1, N_2) and one Uniform distribution (U), from which we sample with 72.5%, 26.5% and 1% respectively. For some constant v:

- We uniformly sample a shift c from $[0, 2 \cdot v]$, that is used to push the centers of N_1 and N_2 to $x > 0$ and $x < 0$ respectively.
- We sample each value from $N_1(c, v)$, $N_2(-c, 3 \cdot v)$, and $U_1(-15 \cdot v, 15 \cdot v)$ randomly, with the weighting given above.

- We normalize (subtract mean, divide by standard deviation) our sampled distribution and then scale it to the desired standard deviation.
- The predictors produce non-smooth distributions. We simulate that by sampling 15 times fewer values as needed, and repeat them as often.

As part of the simulation, the code for the mixed distribution was made available. Figure A.5 shows that the resulting simulated deviation distributions generally resemble a common predictor pattern. We do not account for differences in predictors, training set sizes or more, since that may become too specific and over-engineered.

Appendix A.7 visualizes simulation sanity checks. We find that the simulation is slightly pessimistic and simplified but resembles the results of actual predictors.

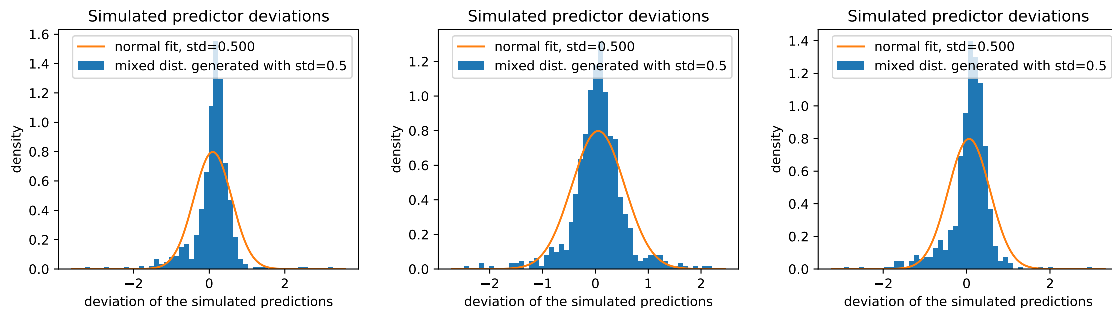


Figure A.5: The sampled values of gaussian+uniform fit the measured predictor mistakes better than a single distribution, as they are roughly normally distributed, but include outliers.

A.6 Limits of MRA

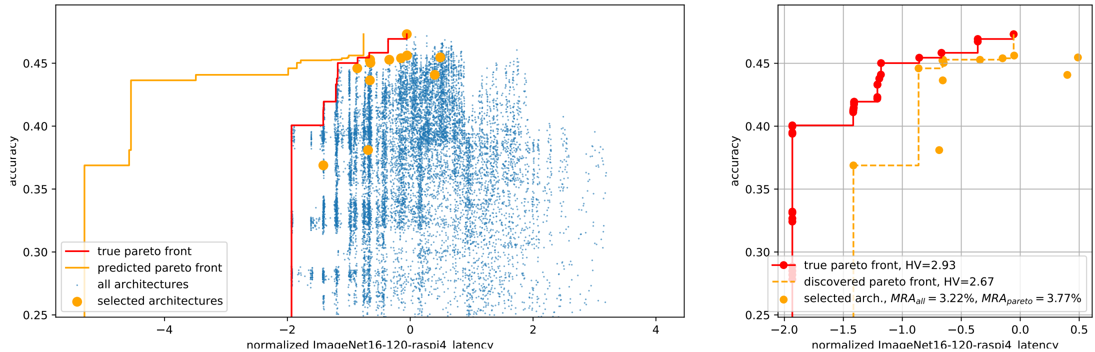


Figure A.6: Similar to Figure 5.5. When the discovered Pareto set is considerably worse than the true Pareto set, it is possible for the Mean Reduction of Accuracy of the Pareto subset (MRA_{pareto}) to be *worse* than the average over all architectures (MRA_{all}). This naturally happens more frequently for worse predictors with a high sampling standard deviation and low KT correlation. Consequentially, the difference between MRA_{all} and MRA_{pareto} is wider for better predictors (see Figure 5.6).
Additionally, all of the selected non-Pareto-front members are clustered in a high-latency area and redundant with each other. This emphasizes the limitations of just considering drops of accuracy, as the hardware metric aspect is ignored. In this case, the predictor-guided selection failed to find a low-latency solution. Hypervolume solves these problems but is a less intuitive metric.

A.7 Simulation sanity check

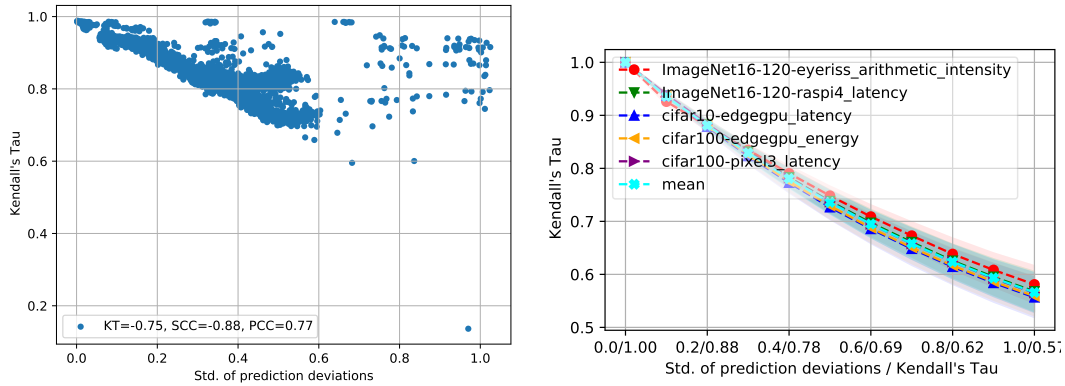


Figure A.7: Standard deviation over the predictor deviations (x axis) and Kendall's Tau correlation (y axis), for the trained predictors on HW-NAS-Bench (left) and in simulation (right). The simulated predictor inaccuracies are slightly pessimistic (low KT) but still match the true values.

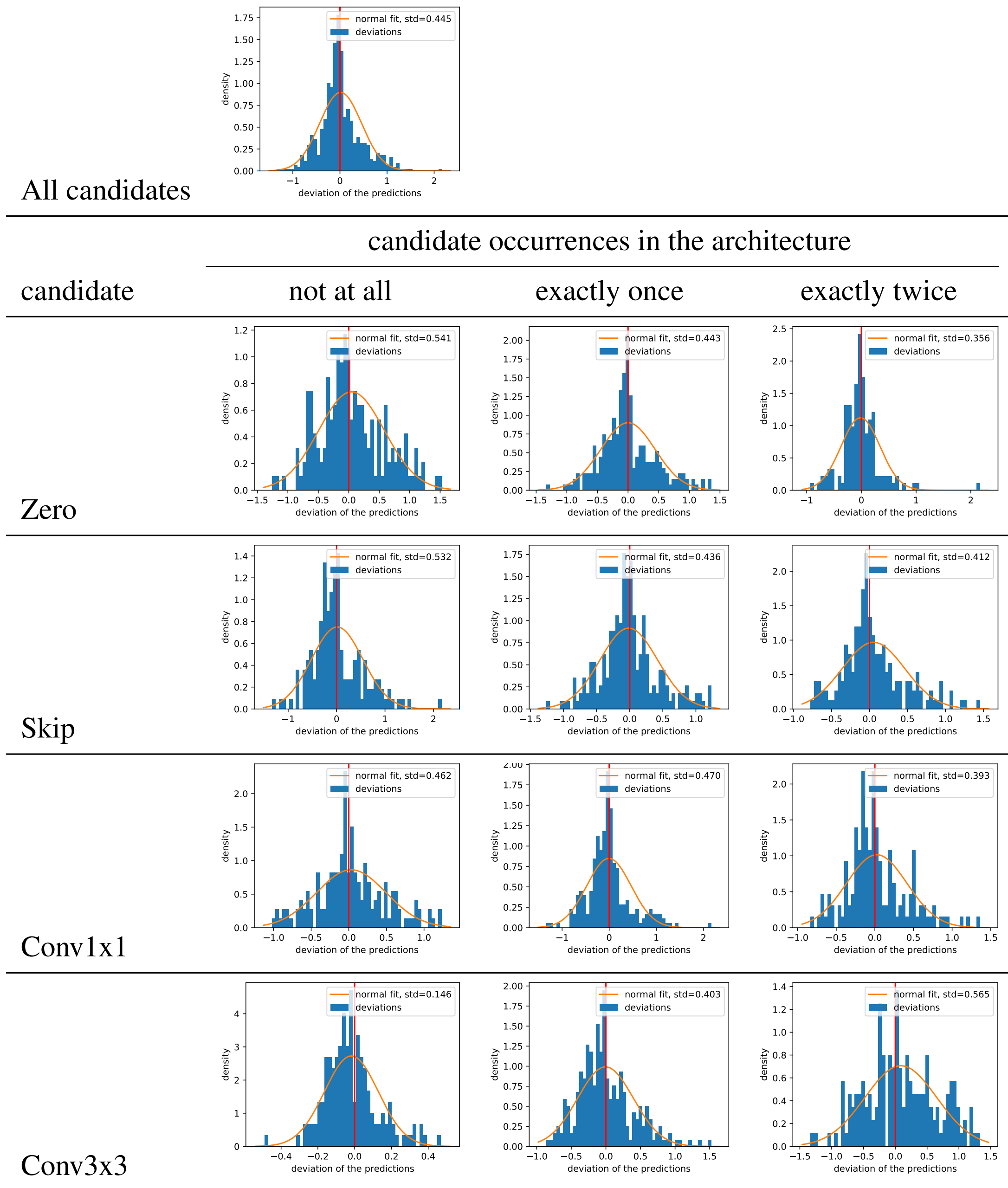


Table A.5: How a trained XGB predictor deviates from the ground-truth values for different architecture subsets, akin to Figure 5.4. While they are not exactly the same, they still resemble the distribution over the entire test set (top plot, 625 samples). One noteworthy exception is when no Conv3x3 operations are used at all, in which case the standard deviation is considerably smaller.

Appendix B

Conditional super-network weights

B.1 Super-network correlations

Table B.1: Kendall's Tau ranking correlation values between super network predictions and ground truth values, computed on all (KT all) and only the top 50 ground truth networks (KT 50). The best column-wise values are marked in bold.
We find that is possible but not certain to improve the ranking correlation over the baseline values, and also not by much. Nonetheless, this results in clearly observable improvement peaks as seen in Section 6.4.1.

	NAS-Bench 201						NAS-Bench-Macro	
	full		no Zero		only Conv.		full	
	KT all	KT 50	KT all	KT 50	KT all	KT 50	KT all	KT 50
baseline	0.56	-0.06	0.46	0.14	0.56	0.31	0.73	0.30
split at 25							0.72	0.34
split at 30							0.73	0.31
split at 35							0.72	0.34
split at 38							0.73	**0.36**
split at 40							0.73	0.31
split at 42							0.73	0.34
split at 45							0.73	0.36
split at 47							**0.74**	0.26
split at 125	0.56	-0.08	0.50	**0.19**	0.54	0.26		
split at 140	0.55	-0.11	0.49	0.13	0.54	0.29		
split at 145	0.56	-0.10	0.50	0.12	0.58	0.34		
split at 150	**0.57**	**-0.04**	**0.51**	0.17	**0.60**	**0.36**		
split at 155	0.56	**-0.04**	**0.51**	0.16	0.56	0.33		
split at 160	**0.57**	-0.07	0.48	**0.19**	0.57	0.32		
split at 175	**0.57**	-0.09	**0.51**	**0.19**	0.56	0.31		
split at 200	0.56	**-0.04**	**0.51**	**0.19**	0.54	0.27		

B.2 Network designs

Tables B.2 and B.3 summarize the super-networks for NAS-Bench-201 and NAS-Bench-Macro, respectively. Both closely follow the design of the original full-sized networks, except for having all candidate operations available. While NAS-Bench-201 networks usually have five cells of shared topology per stage, our super-network uses only two. This is a typical choice to increase the search efficiency (Zoph *et al.* (2018); Real *et al.* (2018); Pham *et al.* (2018); Liu *et al.* (2019) and more, see Chapter 2.4.1).

	input size		
cell index	channels	spatial	params
stem	3	32×32	464
0	16	32×32	16,896
1	16	32×32	16,896
2	16	32×32	14,464
3	32	16×16	67,584
4	32	16×16	67,584
5	32	16×16	57,600
6	64	8×8	270,336
7	64	8×8	270,336
head	64	8×8	778
sum			782,938

Table B.2: NAS-Bench 201 super-network without additional weights on CIFAR10, using two cells per stage (cells 2 and 5 are fixed reduction cells). As the cell topologies are shared, only six operation choices exist.

	input size		
layer index	channels	spatial	params
stem	3	32×32	928
0	32	32×32	36,896
1	64	16×16	87,616
2	64	16×16	133,184
3	128	8×8	322,688
4	128	8×8	322,688
5	128	8×8	503,936
6	256	4×4	1,235,200
7	256	4×4	1,235,200
head	256	4×4	343,050
sum			4,221,386

Table B.3: NAS-Bench-Macro super-network without additional weights on CIFAR10. In each of the 8 layers, one of the three available operations is chosen for the final architecture.

Table B.4: Training details of super-networks and the evaluation networks for all NAS-Bench 201 and NAS-Bench-Macro networks. Due to using the same method on the same dataset (Single-Path One-Shot on CIFAR10), the configurations are almost identical. If details are not mentioned (such as gradient clipping or dropout), they have not been used.

	NAS-Bench 201	NAS-Bench-Macro
Optimizer	SGD	SGD
initial learning rate	0.025	0.025
final learning rate	1e-5	1e-5
learning rate decay	cosine	cosine
momentum	0.9	0.9
weight decay	3e-4	3e-4
weight decay applies to BatchNorm	no	no
epochs	250	50
data input shape	3×32×32	3×32×32
batch size	256	256
training augmentations		
pixel shift	4	4
random horizontal flipping	yes	yes
normalization	yes	yes
evaluation augmentations		
normalization	yes	yes

B.3 Training and evaluation details

The baseline super-networks (without additional weights) are outlined in Appendix B.2. These networks are trained following exactly the same protocol, except for the optional weight splitting in specific epochs. The most important details are summarized in Table B.4. The training protocols loosely follow DARTS (Liu *et al.*, 2019). All models are trained on CIFAR10 (Krizhevsky *et al.*, 2009), where 5000 images are withheld for validation. All experiments were performed using PyTorch 1.7.0 on Nvidia 1080 Ti GPUs with driver version 440.64, CUDA 10.2, CuDNN 7605, and random seeds $\{0, ..., 9\}$.

It is also noteworthy that the BatchNorm statistics of every architecture within the super-network have to be adjusted by performing 20 forward passes (without computing gradients) just prior to evaluating that specific architecture. This is a standard routine of SPOS (Guo *et al.*, 2020) which significantly improves the ranking correlation. However, we did not reset the statistics entirely before the evaluation, which we found detrimental in nearly all tested cases.

Appendix C

UniNAS

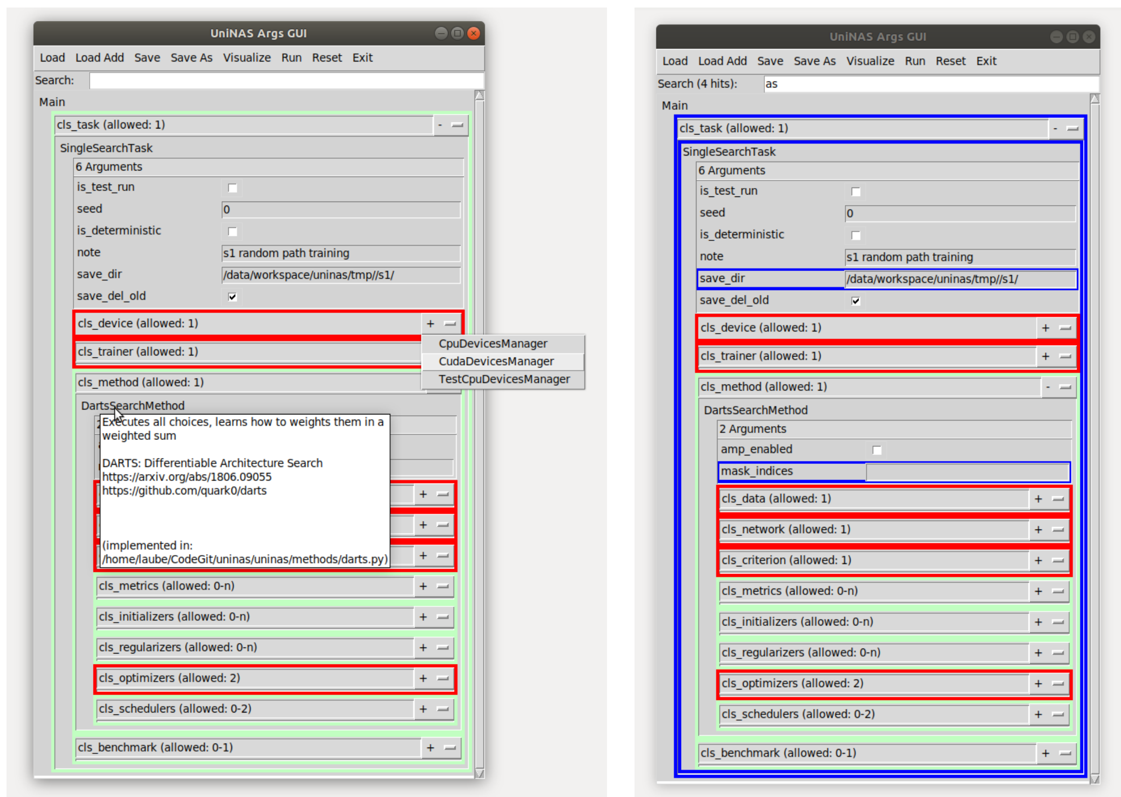


Figure C.1: Additional images for the graphical user interface (GUI).
Left: Hovering the mouse cursor over any name brings up a tooltip, describing the comment in the code and in which file it is implemented. Pressing the Plus and Minus dropdown buttons on the right side enables adding and removing any appropriate classes in the tree structure.
Right: By adding a search text (top), matches are highlighted in blue. The text "as" can be present in argument names ("cls_**task**", "m**ask**_indices"), module names ("SingleSearch**Task**"), or argument values ("save_dir" has ".../unin**as**/...").

```
>--------------------------------------------- Args ---------------------------------------------<
 > cls_task                               SingleSearchTask                                  task
 > cls_device                             CudaDevicesManager                                device manager
 > cls_trainer                            SimpleTrainer                                     trainer
 > cls_method                             UniformRandomMethod                               method
 > cls_benchmark                                                                            immediately look up the search result in this [...]
 > cls_callbacks                          CheckpointCallback                                training callbacks
 > cls_clones                                                                               training clones
 > cls_exp_loggers                        TensorBoardExpLogger                              experiment logger
 > cls_data                               Cifar10Data                                       data set
 > cls_network                            SearchUninasNetwork                               network
 > cls_criterion                          CrossEntropyCriterion                             criterion
 > cls_metrics                            AccuracyMetric                                    training metric
 > cls_initializers                                                                         weight initializer
 > cls_regularizers                       DropOutRegularizer                                regularizer
 > cls_optimizers                         SGDOptimizer                                      optimizer
 > cls_schedulers                         CosineScheduler                                   scheduler
 > cls_augmentations                      DartsCifarAug                                     data augmentation
 > cls_network_body                       StackedCellsNetworkBody                           network
 > cls_network_stem                       ConvStem                                          network stem
 > cls_network_heads                      Bench201Head                                      network heads
 > cls_network_cells                      Bench201CNNSearchCell, Bench201ReductionCell      network cells
 > cls_network_cells_primitives           Bench201Primitives, Bench201Primitives            network cells primitives
 > SingleSearchTask.is_test_run           True                                              test runs stop epochs early
 > SingleSearchTask.seed                  0                                                 random seed for the experiment
 > SingleSearchTask.is_deterministic      False                                             use deterministic operations
 > SingleSearchTask.note                  s1 SPOS-like training                             just to take notes
 > SingleSearchTask.save_dir              /tmp/demo/icw/train_supernet/                     where to save
 > SingleSearchTask.save_del_old          True                                              wipe the save dir before starting
 > CudaDevicesManager.num_devices         1                                                 number of available devices
 > CudaDevicesManager.use_cudnn           True                                              try using cudnn
 > CudaDevicesManager.use_cudnn_benchmark True                                              use cudnn benchmark
 > SimpleTrainer.max_epochs               10                                                max training epochs, affects schedulers + regularizers
 > SimpleTrainer.stop_epoch               -1                                                stop after training n epochs anyway, if > 0
 > SimpleTrainer.log_fs                   False                                             log file system usage
 > SimpleTrainer.log_ram                  False                                             log RAM usage
 > SimpleTrainer.log_device               True                                              log device usage
 > SimpleTrainer.eval_last                10                                                run eval for the last n epochs, always if <0
 > SimpleTrainer.test_last                10                                                run test for the last n epochs, always if <0
 > SimpleTrainer.accumulate_batches       1                                                 accumulate gradients over n batches before stepping [...]
 ...
 > StackedCellsNetworkBody.cell_order     n, n, r, n, n, r, n, n                            arrangement of cells
 > ConvStem.features                      16                                                num output features of this stem
 ...

>---------------------------------------- setting up... -----------------------------------------<
Data Set: splitting the training set, will use 5000 data points as validation set
Building StackedCellsNetworkBody:
    cell index         name    class             input shapes            output shapes          #params
                       -       ConvStem          Shape(3, 32, 32)        [Shape(16, 32, 32)]    464
    0                  n       Bench201CNNCell   [Shape(16, 32, 32)]     [Shape(16, 32, 32)]    18160
    1                  n       Bench201CNNCell   [Shape(16, 32, 32)]     [Shape(16, 32, 32)]    18160
    2                  r       SingleLayerCell   [Shape(16, 32, 32)]     [Shape(32, 16, 16)]    14464
    3                  n       Bench201CNNCell   [Shape(32, 16, 16)]     [Shape(32, 16, 16)]    67040
    4                  n       Bench201CNNCell   [Shape(32, 16, 16)]     [Shape(32, 16, 16)]    67040
    5                  r       SingleLayerCell   [Shape(32, 16, 16)]     [Shape(64, 8, 8)]      57600
    6                  n       Bench201CNNCell   [Shape(64, 8, 8)]       [Shape(64, 8, 8)]      256960
    7                  n       Bench201CNNCell   [Shape(64, 8, 8)]       [Shape(64, 8, 8)]      256960
                       -       Bench201Head      Shape(64, 8, 8)         Shape(10)              778
    complete network                             Shape(3, 32, 32)        [Shape(10)]            757626
Network built, it has 757626 parameters!
Using device: CudaDeviceMover([0])
Continuously logging (devices=CudaDeviceMover([0]), RAM=False, file_system=False) each 5s
...

>--------------------------------------- Weight strategy ----------------------------------------<
RandomChoiceStrategy("default", 6 architecture weights)
Weights:
   name               num choices    used
 > n/block-0/1/op-0   5              6x
 > n/block-1/2/op-0   5              6x
 > n/block-1/2/op-1   5              6x
 > n/block-2/3/op-0   5              6x
 > n/block-2/3/op-1   5              6x
 > n/block-2/3/op-2   5              6x
```

Figure C.2: Excerpts of UniNAS' text output. **Top**: The names, values, and help text of all (meta-) arguments. The effect of the last two can be observed in the network structure. **Center**: Since the network code is well-defined, it is possible to generate an overview of layers, inputs, outputs, and parameters. **Bottom**: The weight strategy can present the interesting information about the used architecture weights. There are five candidates in the chosen operation set (*Bench201Primitives*), and six cells "n" with shared architecture.

Figure C.3: The full argument tree to train a FairNAS-like super-network, extending Figure 7.2. The model is trained on ImageNet using the CrossEntropy loss, SGD and a cosine learning rate schedule. The tracked accuracy is used for checkpointing, and dropout is enabled. The different cell definition model has six stages with defined numbers of channels (32, 40, 80, 96, 192, 320), the last cell definition retains the current input and output sizes.

```
1  @Register.network_mixed_op()
2  class MixedOp(SumParallelModules):
3    def __init__(self, submodules: list, name, strategy_name):
4      # store all arguments, thus including them in the config
5      super().__init__(submodules)
6      self._add_to_kwargs(name=name, strategy_name=strategy_name)
7
8      # create the needed architecture weights
9      self.sm = StrategyManager()  # singleton class
10     self.ws = self.sm.make_weight(strategy_name, name, submodules)
11
12   def forward(self, x: torch.Tensor) -> torch.Tensor:
13     # let the weight strategy decide how to forward inputs
14     return self.ws.combine(self.name, x, self.submodules)
15
16   def config(self, finalize=True, **_) -> dict:
17     # describe this module, so that it can be rebuilt later
18     if finalize:
19       # only a subset of the candidates are requested
20       # ask the weight strategy which candidates are best
21       indices = self.ws.get_finalized_indices(self.name)
22       modules = [self.submodules[i] for i in indices]
23       if len(modules) == 1:
24         return modules[0].config(finalize, **_)
25       return SumParallelModules(modules).config(finalize, **_)
26     else:
27       # the entire super-network is requested
28       return super().config(finalize=finalize, **_)
29
30   @classmethod
31   def from_config(cls, **kwargs):
32     # the rebuilding of owned code sub-modules is omitted
33     # the global register is used to create Modules by name
34     submodules_ = ...
35     submodule_lists_ = ...
36     submodule_dicts_ = ...
37     # rebuild this module with the exact same arguments as before
38     return cls(**submodules_, **submodule_lists_, **
                                         submodule_dicts_, **kwargs)
```

Figure C.4: Excerpt of the UniNAS MixedOp code, an operation that manages multiple candidate operations. They are stored (Lines 5 and 6) and used in a forward pass (Line 14). The methods starting in Lines 16 and 31 define how this MixedOp module is exported as a JSON description and later rebuilt from such. The *from_config* function belongs to a super-class that every UniNAS network module inherits from and is not required to be implemented again in any new class. It is only displayed for completeness.

Symbols

α_i	architecture weight of operation o_i
$\mathcal{A}$	set of all possible architectures, architecture space
B	number of blocks in a cell
$\mathcal{D}$	data set with pairs of inputs and targets $\{(x_1, y_1), ..., (x_N, y_N)\}$
f and f_p	in the context of architecture metric prediction: true measurements and predictor-based estimates, respectively
$\mathcal{L}$	objective function, loss function, error function
o_i	candidate operation o at edge i (Chapter 4)
$O_{(i,j)}$	candidate operation number j at layer/edge i (Chapter 6)
$\mathcal{O}$	set of all candidate operations
ρ	Spearman's rank correlation coefficient
τ	Kendall's Tau, a ranking correlation metric
θ	parameters of a network
ω	parameters of the controller that samples networks
Ω	training configuration. Includes optimizer, schedule, network design, regularization, ...

Abbreviations

AutoML	Automated Machine Learning
DARTS	Differentiable Architecture Search (Liu *et al.*, 2019)
DAG	Directed Acyclic Graph
EA	Evolutionary Algorithm
ENAS	Efficient Neural Architecture Search (Pham *et al.*, 2018)
GUI	Graphical User Interface
HPO	Hyper-Parameter Optimization
HV	Hypervolume, the area or volume under the Pareto front
KT	Kendall's Tau, a ranking correlation metric
ML	Machine Learning
MRA	Mean Reduction of Accuracy
NAS	Neural Architecture Search
OFA	Once-For-All (Cai *et al.*, 2020)
PR-DARTS	Prune and Replace DARTS, PR-NAS based on DARTS
PR-NAS	Prune and Replace NAS (Laube and Zell, 2019a)
RL	Reinforcement Learning
SCC	Spearman's rank correlation coefficient
SGD	Stochastic Gradient Descent
SPOS	Single Path One-Shot (Guo *et al.*, 2020)

Bibliography

A. Buslaev, A. Parinov, E. K. V. I. I. and Kalinin, A. A. (2018). Albumentations: fast and flexible image augmentations. *ArXiv e-prints*.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283. Software available from tensorflow.org.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, **5**(1), 54–65.

Bai, J., Lu, F., Zhang, K., *et al.* (2019). ONNX: Open Neural Network Exchange. `https://github.com/onnx/onnx`.

Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. (2017). Neural Optimizer Search with Reinforcement Learning. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 459–468. PMLR.

Bender, G., Liu, H., Chen, B., Chu, G., Cheng, S., Kindermans, P.-J., and Le, Q. (2020). Can weight sharing outperform random architecture search? An investigation with TuNAS. `https://github.com/google-research/google-research/tree/master/tunas`.

Benmeziane, H., Maghraoui, K. E., Ouarnoughi, H., Niar, S., Wistuba, M., and Wang, N. (2021). A Comprehensive Survey on Hardware-Aware Neural Architecture Search. *CoRR*, **abs/2101.09336**.

Bishop, C. M. (2007). *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer.

Blank, J. and Deb, K. (2020). pymoo: Multi-Objective Optimization in Python. *IEEE Access*, **8**, 89497–89509.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2018). SMASH: One-Shot Model Architecture Search through HyperNetworks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Cai, H., Yang, J., Zhang, W., Han, S., and Yu, Y. (2018). Path-Level Network Transformation for Efficient Architecture Search. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 677–686. PMLR.

Cai, H., Zhu, L., and Han, S. (2019). ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. (2020). Once-for-All: Train One Network and Specialize it for Efficient Deployment.

Candela, J. Q. and Rasmussen, C. E. (2005). A Unifying View of Sparse Approximate Gaussian Process Regression. *J. Mach. Learn. Res.*, **6**, 1939–1959.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

Chen, T., Goodfellow, I. J., and Shlens, J. (2016). Net2Net: Accelerating Learning via Knowledge Transfer. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Chen, X., Xie, L., Wu, J., and Tian, Q. (2019). Progressive Differentiable Architecture Search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1294–1303.

Chrabaszcz, P., Loshchilov, I., and Hutter, F. (2017). A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*.

Chu, X., Zhang, B., Xu, R., and Li, J. (2019a). FairNAS: Rethinking Evaluation Fairness of Weight Sharing Neural Architecture Search. *CoRR*, **abs/1907.01845**.

Chu, X., Zhang, B., Li, J., Li, Q., and Xu, R. (2019b). ScarletNAS: Bridging the Gap Between Scalability and Fairness in Neural Architecture Search. *CoRR*, **abs/1908.06022**.

Chu, X., Zhou, T., Zhang, B., and Li, J. (2020a). Fair DARTS: Eliminating Unfair Advantages in Differentiable Architecture Search. In *European Conference on Computer Vision*, pages 465–480. Springer.

Chu, X., Li, X., Lu, Y., Zhang, B., and Li, J. (2020b). MixPath: A Unified Approach for One-shot Neural Architecture Search. *CoRR*, **abs/2001.05887**.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, **20**(3), 273–297.

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. (2018). AutoAugment: Learning Augmentation Policies from Data.

Dai, X., Wan, A., Zhang, P., Wu, B., He, Z., Wei, Z., Chen, K., Tian, Y., Yu, M., Vajda, P., and Gonzalez, J. E. (2020). FBNetV3: Joint Architecture-Recipe Search using Neural Acquisition Function. *CoRR*, **abs/2006.02049**.

Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, **6**(2), 182–197.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Devries, T. and Taylor, G. W. (2017). Improved Regularization of Convolutional Neural Networks with Cutout. *CoRR*, **abs/1708.04552**.

Dong, J., Cheng, A., Juan, D., Wei, W., and Sun, M. (2018). DPP-Net: Device-Aware Progressive Search for Pareto-Optimal Neural Architectures. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XI*, volume 11215 of *Lecture Notes in Computer Science*, pages 540–555. Springer.

Dong, X. and Yang, Y. (2019). Searching for a Robust Neural Architecture in Four GPU Hours. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 1761–1770. Computer Vision Foundation / IEEE.

Dong, X. and Yang, Y. (2020). NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Duan, T., Anand, A., Ding, D. Y., Thai, K. K., Basu, S., Ng, A. Y., and Schuler, A. (2020). Ngboost: Natural gradient boosting for probabilistic prediction. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 2690–2700. PMLR.

Duan, Y., Chen, X., Xu, H., Chen, Z., Liang, X., Zhang, T., and Li, Z. (2021). TransNAS-Bench-101: Improving Transferability and Generalizability of Cross-Task Neural Architecture Search. *CoRR*, **abs/2105.11871**.

Elsken, T., Metzen, J. H., and Hutter, F. (2018). Simple and efficient architecture search for Convolutional Neural Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Neural Architecture Search: A Survey. *J. Mach. Learn. Res.*, **20**, 55:1–55:21.

Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. (2020). Single Path One-Shot Neural Architecture Search with Uniform Sampling. In *European Conference on Computer Vision*, pages 544–560. Springer.

Han, D., Kim, J., and Kim, J. (2017). Deep Pyramidal Residual Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6307–6315. IEEE Computer Society.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Comput.*, **9**(8), 1735–1780.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision applications. *CoRR*, **abs/1704.04861**.

Hu, J., Shen, L., and Sun, G. (2018). Squeeze-and-Excitation Networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141.

Hu, S., Xie, S., Zheng, H., Liu, C., Shi, J., Liu, X., and Lin, D. (2020). DSNAS: Direct Neural Architecture Search without Parameter Retraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12084–12092.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2017). Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society.

Hundt, A., Jain, V., and Hager, G. D. (2019). sharpDARTS: Faster and More Accurate Differentiable Architecture Search. *CoRR*, **abs/1903.09900**.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR.

Jiajin, Z. (2020). Huawei Noah Vega. `https://github.com/huawei-noah/vega/`.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3146–3154.

Kingma, D. P. and Ba, J. (2015). Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. **4**(4).

Krizhevsky, A., Nair, V., and Hinton, G. (2009). CIFAR-10 (Canadian Institute for Advanced Research).

Larsson, G., Maire, M., and Shakhnarovich, G. (2017). FractalNet: Ultra-Deep Neural Networks without Residuals. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Laube, K. A. (2019). Prune and Replace NAS GitHub. `https://github.com/cogsys-tuebingen/prdarts`.

Laube, K. A. (2021). The UniNAS framework: combining modules in arbitrarily complex configurations with argument trees. `https://github.com/cogsys-tuebingen/uninas`, `https://arxiv.org/abs/2112.01796`.

Laube, K. A. and Zell, A. (2019a). Prune and Replace NAS. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 915–921. IEEE.

Laube, K. A. and Zell, A. (2019b). ShuffleNASNets: Efficient CNN models through modified Efficient Neural Architecture Search. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, pages 1–6. IEEE.

Laube, K. A. and Zell, A. (2021a). Exploring single-path architecture search ranking correlations. `https://openreview.net/forum?id=J40FkbdldTX`.

Laube, K. A. and Zell, A. (2021b). Inter-choice dependent super-network weights. *CoRR*, **abs/2104.11522**.

Laube, K. A., Mutschler, M., and Zell, A. (2022). What to expect of hardware metric predictors in NAS. In *AutoML-Conf 2022*.

Lee, H., Lee, S., Chong, S., and Hwang, S. J. (2021). HELP: Hardware-Adaptive Efficient Latency Predictor for NAS via Meta-Learning. *CoRR*, **abs/2106.08630**.

Li, C., Peng, J., Yuan, L., Wang, G., Liang, X., Lin, L., and Chang, X. (2020). Blockwisely Supervised Neural Architecture Search with Knowledge Distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1989–1998.

Li, C., Yu, Z., Fu, Y., Zhang, Y., Zhao, Y., You, H., Yu, Q., Wang, Y., and Lin, Y. (2021a). HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark. *CoRR*, **abs/2103.10584**.

Li, G., Mandal, S. K., Ogras, Ü. Y., and Marculescu, R. (2021b). FLASH: Fast Neural Architecture Search with Hardware Optimization. *CoRR*, **abs/2108.00568**.

Li, L. and Talwalkar, A. (2020). Random Search and Reproducibility for Neural Architecture Search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR.

Liang, H., Zhang, S., Sun, J., He, X., Huang, W., Zhuang, K., and Li, Z. (2019). DARTS+: Improved Differentiable Architecture Search with Early Stopping. *CoRR*, **abs/1909.06035**.

Liaw, A., Wiener, M., *et al.* (2002). Classification and Regression by randomForest. *R news*, **2**(3), 18–22.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollar, P., and Zitnick, L. (2014). Microsoft COCO: Common Objects in Context. In *ECCV*. European Conference on Computer Vision.

Lindauer, M. and Hutter, F. (2020). Best Practices for Scientific Research on Neural Architecture Search. *Journal of Machine Learning Research*, **21**(243), 1–18.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018). Progressive Neural Architecture Search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable Architecture Search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. (2017). Learning Efficient Convolutional Networks through Network Slimming. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2755–2763. IEEE Computer Society.

Loshchilov, I. and Hutter, F. (2017). SGDR: Stochastic Gradient Descent with Warm Restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Lu, Z., Sreekumar, G., Goodman, E., Banzhaf, W., Deb, K., and Boddeti, V. N. (2020). Neural Architecture Transfer.

Luo, R., Tian, F., Qin, T., Chen, E., and Liu, T. (2018). Neural Architecture Optimization. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7827–7838.

Ma, N., Zhang, X., Zheng, H., and Sun, J. (2018). ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIV*, volume 11218 of *Lecture Notes in Computer Science*, pages 122–138. Springer.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.*, **19**(2), 313–330.

Mellor, J., Turner, J., Storkey, A., and Crowley, E. J. (2020). Neural Architecture Search without Training.

Mendoza, D. M. and Wang, S. (2020). Predicting Latency of Neural Network Inference.

Nayman, N., Aflalo, Y., Noy, A., and Zelnik, L. (2021). HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7979–7990. PMLR.

Noy, A., Nayman, N., Ridnik, T., Zamir, N., Doveh, S., Friedman, I., Giryes, R., and Zelnik, L. (2020). ASAP: Architecture Search, Anneal and Prune. In S. Chiappa and R. Calandra, editors, *The 23rd International Conference on Artificial Intelligence and*

Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy], volume 108 of *Proceedings of Machine Learning Research*, pages 493–503. PMLR.

Papert, S. (1966). The summer vision project. *MIT library*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., *et al.* (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research*, **12**(Oct), 2825–2830.

Peng, D., Dong, X., Real, E., Tan, M., Lu, Y., Bender, G., Liu, H., Kraft, A., Liang, C., and Le, Q. (2020a). Pyglove: Symbolic programming for automated machine learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Peng, H., Du, H., Yu, H., Li, Q., Liao, J., and Fu, J. (2020b). Cream of the crop: Distilling prioritized paths for one-shot neural architecture search. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4092–4101. PMLR.

Ponomarev, E., Matveev, S. A., and Oseledets, I. V. (2020). LETI: Latency Estimation Tool and Investigation of Neural Networks inference on Mobile GPU. *CoRR*, **abs/2010.02871**.

Ramachandran, P., Zoph, B., and Le, Q. V. (2018). Searching for Activation Functions. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net.

Rasmussen, C. E. (2003). Gaussian Processes in Machine Learning. In O. Bousquet, U. von Luxburg, and G. Rätsch, editors, *Advanced Lectures on Machine Learning,*

ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003, Revised Lectures, volume 3176 of *Lecture Notes in Computer Science*, pages 63–71. Springer.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized Evolution for Image Classifier Architecture Search.

Real, E., Liang, C., So, D. R., and Le, Q. V. (2020). AutoML-Zero: Evolving Machine Learning Algorithms From Scratch. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR.

Ruchte, M., Zela, A., Siems, J., Grabocka, J., and Hutter, F. (2020). Naslib: A modular and flexible neural architecture search library. `https://github.com/automl/NASLib`.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520.

Saunders, C., Gammerman, A., and Vovk, V. (1998). Ridge Regression Learning Algorithm in Dual Variables. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, page 515–521, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Sciuto, C., Yu, K., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the Search Phase of Neural Architecture Search. *CoRR*, **abs/1902.08142**.

Shah, S., Dey, D., Majumdar, S., and Sagar, D. (2020). Archai. `https://github.com/microsoft/archai`.

Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J. T., and Zhang, T. (2019). Bridging the Gap between Sample-based and One-shot Neural Architecture Search with BONAS. *arXiv preprint arXiv:1911.09336*.

Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J. T., and Zhang, T. (2020). Bridging the Gap between Sample-based and One-shot Neural Architecture Search with BONAS. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Siems, J., Zimmer, L., Zela, A., Lukasik, J., Keuper, M., and Hutter, F. (2020). NAS-Bench-301 and the Case for Surrogate Benchmarks for Neural Architecture Search.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. (2016). Bayesian Optimization with Robust Bayesian Neural Networks. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4134–4142.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, **15**(1), 1929–1958.

Stamoulis, D., Ding, R., Wang, D., Lymberopoulos, D., Priyantha, B., Liu, J., and Marculescu, D. (2019). Single-Path NAS: Designing Hardware-Efficient ConvNets in less than 4 Hours. In *arXiv preprint arXiv:1904.02877*.

Su, X., Huang, T., Li, Y., You, S., Wang, F., Qian, C., Zhang, C., and Xu, C. (2021). Prioritized Architecture Sampling With Monto-Carlo Tree Search. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 10968–10977. Computer Vision Foundation / IEEE.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. pages 1–9.

Tan, M. and Le, Q. V. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *CoRR*, **abs/1905.11946**.

Tan, M. and Le, Q. V. (2021). EfficientNetV2: Smaller Models and Faster Training. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 10096–10106. PMLR.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. (2019). MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2820–2828. Computer Vision Foundation / IEEE.

Tan, M., Pang, R., and Le, Q. V. (2020). Efficientdet: Scalable and efficient object detection. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10778–10787. Computer Vision Foundation / IEEE.

Tenorio, M. F. and Lee, W.-T. (1988). Self organizing neural networks for the identification problem. pages 57–64.

Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., Vajda, P., and Gonzalez, J. E. (2020). FBNetV2: Differentiable Neural Architecture Search for Spatial and Channel Dimensions. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 12962–12971. IEEE.

Wang, D., Li, M., Gong, C., and Chandra, V. (2021a). AttentiveNAS: Improving Neural Architecture Search via Attentive Sampling. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 6418–6427. Computer Vision Foundation / IEEE.

Wang, R., Chen, X., Cheng, M., Tang, X., and Hsieh, C. (2021b). RANK-NOSH: Efficient Predictor-Based Architecture Search via Non-Uniform Successive Halving. *CoRR*, **abs/2108.08019**.

Wei, T., Wang, C., Rui, Y., and Chen, C. W. (2016). Network Morphism. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 564–572. JMLR.org.

Wen, W., Liu, H., Chen, Y., Li, H. H., Bender, G., and Kindermans, P. (2020). Neural Predictor for Neural Architecture Search. In A. Vedaldi, H. Bischof, T. Brox, and J. Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXIX*, volume 12374 of *Lecture Notes in Computer Science*, pages 660–676. Springer.

Wess, M., Ivanov, M., Unger, C., Nookala, A., Wendt, A., and Jantsch, A. (2021). ANNETTE: Accurate Neural Network Execution Time Estimation With Stacked Models. *IEEE Access*, **9**, 3545–3556.

White, C., Neiswanger, W., and Savani, Y. (2019). BANANAS: Bayesian Optimization with Neural Architectures for Neural Architecture Search. *arXiv preprint arXiv:1910.11858*.

White, C., Nolen, S., and Savani, Y. (2020). Local Search is State of the Art for Neural Architecture Search Benchmarks.

White, C., Zela, A., Ru, B., Liu, Y., and Hutter, F. (2021). How Powerful are Performance Predictors in Neural Architecture Search? *CoRR*, **abs/2104.01177**.

Wightman, R. (2019). Pytorch image models. `https://github.com/rwightman/pytorch-image-models`.

Wikipedia (2021). Pearson, Spearman and Kendall's Tau. `https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient` and `https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient`.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.*, **8**, 229–256.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. (2019). FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 10734–10742. Computer Vision Foundation / IEEE.

Xie, S., Zheng, H., Liu, C., and Lin, L. (2018). SNAS: Stochastic Neural Architecture Search.

Xu, Y., Xie, L., Zhang, X., Chen, X., Shi, B., Tian, Q., and Xiong, H. (2020). Latency-Aware Differentiable Neural Architecture Search. *CoRR*, **abs/2001.06392**.

Yang, A., Esperança, P. M., and Carlucci, F. M. (2019). Nas evaluation is frustratingly hard. *arXiv preprint arXiv:1912.12522*.

Yang, T., Howard, A. G., Chen, B., Zhang, X., Go, A., Sandler, M., Sze, V., and Adam, H. (2018). NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part X*, volume 11214 of *Lecture Notes in Computer Science*, pages 289–304. Springer.

Yao, Q., Xu, J., Tu, W., and Zhu, Z. (2019). Differentiable Neural Architecture Search via Proximal Iterations. *CoRR*, **abs/1905.13577**.

Yao, S., Zhao, Y., Shao, H., Liu, S., Liu, D., Su, L., and Abdelzaher, T. (2018). Fast-DeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291.

Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., and Hutter, F. (2019). NAS-Bench-101: Towards Reproducible Neural Architecture Search. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114. PMLR.

Yu, J., Jin, P., Liu, H., Bender, G., Kindermans, P., Tan, M., Huang, T. S., Song, X., Pang, R., and Le, Q. (2020a). BigNAS: Scaling up Neural Architecture Search with Big Single-Stage Models.

Yu, K., Ranftl, R., and Salzmann, M. (2020b). How to Train Your Super-Net: An Analysis of Training Heuristics in Weight-Sharing NAS. *CoRR*, **abs/2003.04276**.

Yu, K., Ranftl, R., and Salzmann, M. (2020c). How to Train Your Super-Net: An Analysis of Training Heuristics in Weight-Sharing NAS.

Zagoruyko, S. and Komodakis, N. (2016). Wide Residual Networks. In R. C. Wilson, E. R. Hancock, and W. A. P. Smith, editors, *Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016*. BMVA Press.

Zela, A., Elsken, T., Saikia, T., Marrakchi, Y., Brox, T., and Hutter, F. (2020). Understanding and Robustifying Differentiable Architecture Search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Zhang, Q. (2019). NNI - Neural Network Intelligence. `https://github.com/microsoft/nni`.

Zhang, X., Zhou, X., Lin, M., and Sun, J. (2018). ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 6848–6856. IEEE Computer Society.

Zhang, Y., Lin, Z., Jiang, J., Zhang, Q., Wang, Y., Xue, H., Zhang, C., and Yang, Y. (2020). Deeper Insights into Weight Sharing in Neural Architecture Search. *CoRR*, **abs/2001.01431**.

Zhao, Y., Wang, L., Tian, Y., Fonseca, R., and Guo, T. (2020). Few-shot Neural Architecture Search. *arXiv preprint arXiv:2006.06863*.

Zheng, X., Ji, R., Tang, L., Zhang, B., Liu, J., and Tian, Q. (2019). Multinomial Distribution Learning for Effective Neural Architecture Search. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 1304–1313. IEEE.

Zhuang, J., Tang, T., Ding, Y., Tatikonda, S., Dvornek, N., Papademetris, X., and Duncan, J. (2020). AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients. *Conference on Neural Information Processing Systems*.

Zoph, B. and Le, Q. V. (2016). Neural Architecture Search with Reinforcement Learning.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710.