Topology Learning for Prediction, Generation, and Robustness in Neural Architecture Search

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Jovita Lukasik
aus Hamburg

Mannheim, 2023

Dekan: Dr. Bernd Lübcke, Universität Mannheim
Referent: Prof. Dr.-Ing. Margret Keuper, Universität Siegen
Korreferent: Prof. Dr. Heiner Stuckenschmidt, Universität Mannheim
Korreferent Prof. Dr. Frank Hutter, Albert-Ludwigs-Universität Freiburg

Tag der mündlichen Prüfung: 13. Juli 2023

Abstract

In recent years, deep learning with Convolutional Neural Networks has become the key for success in computer vision tasks. However, designing new architectures is compute-intensive and a tedious trial-and-error process, which depends on human expert knowledge. Neural Architecture Search (NAS) builds on this problem by automating the architecture design process to find high-performing architectures. Yet, initial approaches in NAS rely on training and evaluating thousands of networks, resulting in compute-intensive search times.

In this thesis, we introduce efficient search methods which overcome the heavy search time. First, we focus on presenting a surrogate model to predict the performance of architectures. Significantly, this surrogate model is able to predict the performance of architectures with a topology, which was not seen during training, i.e., our proposed model can extrapolate into unseen regions.

In the second part, we introduce two generative architecture search approaches. The first one is based on a variational autoencoder, which enables to search for architectures directly in the generated latent space, with the ability to generate the found architectures back to its discrete architecture topology. The second approach improves on the former and facilitates a simple generation model, which is furthermore coupled with a surrogate model to search for architectures directly. In addition, we optimize the latent space itself for a direct generation of high-performing architectures.

The third part of this thesis analyzes the widely used differentiable one-shot method DARTS, with the questions, is this method indeed an efficient search method, and how sensitive is this method to domain shifts, hyperparameters, and initializations?

Lastly, we pave the way for robustness in NAS research. We introduce a dataset for architecture design and robustness, which evaluates one complete NAS search space against adversarial attacks and corruptions and thus allows for an in-depth analysis of the architectural design to improve its robustness only by its topology.

Zusammenfassung

In den letzten Jahren hat sich das Deep Learning mit gefalteten neuronalen Netzwerken zum Schlüssel für den Erfolg bei Computer-Vision-Aufgaben entwickelt. Der Entwurf neuer Architekturen ist jedoch rechenintensiv und ein langwieriger Versuch und Irrtum Prozess, der von menschlichem Expertenwissen abhängt. Neuronale Netzwerk Suche (NAS) geht dieses Problem an, indem es den Architekturentwurfsprozess automatisiert, um hochperformante Architekturen zu finden. Die ersten NAS-Ansätze basieren jedoch auf dem Training und der Evaluierung tausender Netzwerke, was zu rechenintensiven Suchzeiten führt.

In dieser Arbeit stellen wir effiziente Suchmethoden vor, die den hohen Suchaufwand überwinden. Zunächst konzentrieren wir uns darauf, ein Surrogatmodell vorzustellen um die Performanz von Architekturen vorherzusagen. Bezeichnenderweise ist dieses Ersatzmodell in der Lage, die Leistung von Architekturen mit einer Topologie vorherzusagen, die beim Training nicht gesehen wurde, d.h. unser vorgeschlagenes Modell kann in ungesehene Regionen extrapolieren.

Im zweiten Teil stellen wir zwei generative Ansätze zur Architektursuche vor. Der erste basiert auf einem Variations-Autoencoder, der es ermöglicht, direkt im generierten latenten Raum nach Architekturen zu suchen und die gefundenen Architekturen in ihre diskrete Architekturtopologie zurück zu generieren. Der zweite Ansatz verbessert den ersten und ist ein einfaches Generierungsmodell, das darüber hinaus mit einem Surrogatmodell gekoppelt ist, um direkt nach Architekturen zu suchen. Darüber hinaus optimieren wir den latenten Raum selbst für eine direkte Generierung von hochperformanten Architekturen.

Der dritte Teil dieser Arbeit analysiert die weit verbreitete differenzierbare One-Shot-Methode DARTS mit den Fragen, ob diese Methode tatsächlich eine effiziente Suchmethode ist und wie empfindlich diese Methode auf Domänenverschiebungen, Hyperparameter und Initialisierungen reagiert.

Schließlich ebnen wir den Weg für die Richtung Robustheit in NAS. Wir führen einen Datensatz für Architekturdesign und Robustheit ein, der einen kompletten NAS-Suchraum gegen gegnerische Angriffe und Verfälschungen evaluiert und somit eine eingehende Analyse des Architekturdesigns ermöglicht, um die Robustheit allein durch die Topologie zu verbessern.

Acknowledgments

Over the past four years, I have come to know and cherish many people from whom I have received tremendous support.

First of all, I would like to express my deepest gratitude to my supervisor Prof. Dr.-Ing. Margret Keuper, especially for giving me the opportunity to do the Ph.D. in the first place. She believed in me from the very beginning, even though I had no informatics background, and has always encouraged me in the course of my doctoral studies. She has always pushed me forward with her ideas and expertise, but also with many discussions. But not to forget are the countless night shifts before deadlines. Ultimately, her invaluable guidance helped me to become the researcher I am today. I am deeply grateful to have been part of her team.

I would also like to thank Prof. Dr. Heiner Stuckenschmidt, with whom I published my first paper, for his time and expertise to review this thesis. I am also immensely thankful for Prof. Dr. Frank Hutter, who is not only the co-examiner of this thesis but also gave me the opportunity to get to know other sides of the research world, like co-organizing a conference workshop or a seminar series, where I am learning every week. I would also like to express my gratitude to Prof. Dr. Bernt Schiele, who, together with Prof. Dr.-Ing. Margret Keuper made it possible for me to learn and grow a lot further in the MPI Informatics environment.

During my doctoral studies, I had the pleasure to work with many people for whom I am very grateful. I would especially like to thank my colleagues at the DWS from the University of Mannheim for the great atmosphere. Although the lunch breaks at 11:30 AM were always very early, they were also fun and provided a great break from the work routine. My special thanks go to Sven, who helped me a lot, especially at the beginning of my studies, to find my way around in the DWS setting and thus made my start much easier. Furthermore, I would like to remember the time when I shared an office with Daniel and his very loud keyboard until Corona divided us so that we are now separated by a thin useless wall since the background noise has not changed at all. I also want to mention my colleagues at the MPI Informatics, even if the time was short, I learned a lot through fruitful discussions and conversations and I am very grateful for this experience.

My special thanks go to my group colleagues Amirhossein, Julia, Kalun, and Steffen with whom I was not only able to have interesting discussions over the last years but also enjoyed spending my time privately, whether in a sneak preview or at dinner. I want to particularly emphasize Steffen's role, from whom I learned a lot and enjoyed all joint cooperations. Furthermore, I would like to thank Shashank, who took the time to read over this thesis. At this point, I would also like to thank my closest friends, who supported me and always cheered me up from the beginning of my studies. Especially, Khadija, who kept me going not only during my master's studies but also during my Ph.D. with her time and lots of coffee. Away from university, Ardita, Kerstin, Marie and Sophie always brightened up my time, especially Sophie and Ardita, with whom I created a great trio in Mannheim, who supported me mentally and most importantly also food-wise.

One of my biggest supporters is and always will be Lukas, who was by my side from the beginning, supporting me every day and getting me back on track when it felt like everything was falling apart. Without him, this thesis probably never would have happened (if only for the template). Thank you for always being there for me and being my rock, and for just hating me sometimes. I would also like to thank Ute and Frank, who always welcomed me with their

generous nature and lots of food. I would also like to thank Hannah, who always took care of me during every visit.

Last but most importantly, I want to thank my parents, Iwona and Zenon, and my brother, David, who have always been there for me and who I can always count on. My biggest goal has always been to make them proud. I have looked up to my brother since I was a little child and always wanted to be a part of everything he was doing, much to his sorrow. In the end, it took us in two different professional directions, but still, he was always a role model for me, and I hope that he can now also be proud of his little sister. Finally, words cannot describe how much I thank my parents for their time and support. They were always there for me with their warmth and their time. They have made me the person I am today and have always reassured me, even in every low, that things will go on and that everything will be fine in the end. This thesis is dedicated to them.

Contents

Abstract **i**

Zusammenfassung **iii**

Acknowledgments **v**

1 Introduction **1**
1.1 Contribution Overview . . . 3
1.1.1 Performance Estimation Strategy . . . 4
1.1.2 Generative Architecture Search . . . 4
1.1.3 One-Shot Architecture Search . . . 6
1.1.4 Robustness of NAS Architectures . . . 6
1.1.5 Excursus on Graph Decomposition . . . 7
1.2 Outline . . . 7
1.3 Publications . . . 10

2 Related Work and Technical Background **11**
2.1 Neural Architecture Search . . . 11
2.1.1 Preliminaries . . . 11
2.1.2 Search Spaces . . . 12
2.1.3 Search Methods . . . 14
2.1.4 One-Shot Methods with Supernets . . . 15
2.1.5 Performance Estimation Strategy . . . 17
2.1.6 Benchmarks . . . 17
2.2 Graph Neural Networks . . . 21
2.2.1 Preliminaries . . . 21
2.3 Technical Background . . . 22
2.3.1 Bayesian Optimization . . . 22
2.3.2 Variational Autoencoder . . . 26

I Performance Estimation Strategy **31**

3 Graph Neural Network-based Prediction Model **33**
3.1 The Graph Encoder . . . 34
3.1.1 Node-Level Propagation . . . 34
3.1.2 Graph-Level Aggregation . . . 35
3.2 Experiments . . . 35
3.2.1 Performance Prediction . . . 36
3.2.2 Training Behavior . . . 38
3.2.3 Comparison to State of the Art . . . 39

3.3 Conclusion . . . 40

II Generative Architecture Search 41

4 Variational Autoencoder-based Graph Embeddings 43
4.1 Related Work . . . 44
4.2 Structural Smooth Graph Autoencoding . . . 45
4.2.1 Encoder . . . 45
4.2.2 Decoder . . . 47
4.2.3 Loss Function and Training . . . 49
4.3 Discussion of the Impact of Isomorphisms . . . 49
4.3.1 Unique Latent Space Representation. . . . 50
4.3.2 Decoding from the Latent Space. . . . 50
4.4 Experiments . . . 51
4.4.1 Autoencoder Abilities. . . . 51
4.4.2 Latent Space Smoothness Observations . . . 53
4.4.3 Performance Prediction from Latent Space . . . 53
4.4.4 Bayesian Optimization . . . 54
4.4.5 Extrapolation Ability . . . 55
4.5 Conclusion . . . 56

5 Generative NAS with Latent Space Optimization 59
5.1 Latent Space Optimization using Weighted Retraining . . . 61
5.1.1 Problem Statement . . . 61
5.1.2 Weighted Retraining . . . 61
5.2 Architecture Generative Model . . . 62
5.2.1 Preliminaries . . . 62
5.2.2 Generative Network . . . 62
5.2.3 Performance Predictor . . . 63
5.2.4 Training Objectives . . . 65
5.2.5 Generative Latent Space Optimization . . . 65
5.2.6 AG-Net Search Process . . . 65
5.3 Experiments . . . 66
5.3.1 Experiments on Tabular Benchmarks . . . 66
5.3.2 Experiments on Surrogate Benchmarks . . . 69
5.3.3 ImageNet Experiments . . . 70
5.3.4 Experiments on Hardware-Aware Benchmark . . . 73
5.3.5 Generator Evaluation . . . 76
5.4 Ablation Studies . . . 76
5.4.1 Ablation on LSO and Backpropagation . . . 76
5.4.2 Oracle Ablation . . . 78
5.4.3 Predictor Ablation – Local Solution . . . 79
5.4.4 Latent Space Ablations . . . 79
5.5 Conclusion . . . 81

III One-Shot Architecture Search **83**

6 Is Differentiable Architecture Search truly a One-Shot Method? **85**
6.1 Differentiable Architecture Search . . . 86
6.2 Related Work . . . 87
6.3 Proposed Search Spaces . . . 88
6.3.1 Sequential Search Space . . . 88
6.3.2 Non-Sequential Search Space . . . 89
6.3.3 Network Operations for Inverse Problems . . . 89
6.4 Evaluating DAS for Inverse Problems . . . 90
6.4.1 Experimental Setup . . . 90
6.4.2 Results . . . 91
6.4.3 Architecture and DAS Performance . . . 93
6.4.4 Improving the Initialization . . . 96
6.4.5 Results Non-sequential Search Space . . . 97
6.5 Conclusions . . . 100

IV Robustness of NAS Architectures **101**

7 A Dataset for Neural Architecture Design and Robustness **103**
7.1 Robustness and Generalization . . . 104
7.2 Related Work . . . 105
7.3 Dataset Generation . . . 106
7.3.1 Architectures in NAS-Bench-201 . . . 106
7.3.2 Robustness to Adversarial Attacks . . . 106
7.3.3 Robustness to Common Corruptions . . . 109
7.4 Use Cases . . . 110
7.4.1 Training-Free Measurements for Robustness . . . 110
7.4.2 NAS on Robustness . . . 112
7.4.3 Analyzing the Effect of Architecture Design on Robustness . . . 113
7.5 Conclusion . . . 117

V Excursus on Graph Clustering **119**

8 Excursus about Clustering on Graphs **121**
8.1 Related Work . . . 122
8.2 Standard Correlation Clustering Formulation . . . 123
8.3 Benders Decomposition for Correlation Clustering . . . 124
8.3.1 Cutting Plane Optimization . . . 127
8.4 Magnanti-Wong Benders Rows . . . 127
8.5 Experiments: Image Segmentation . . . 128
8.6 Conclusions . . . 131

9 Conclusion **133**

9.1 Summary . . . 133
9.2 Outlook and Future Work . . . 134

Appendices **137**

A Search Space Representation **139**
A.1 NAS-Bench-101 . . . 139
A.2 NAS-Bench-201 . . . 139
A.3 DARTS Search Space . . . 139
A.4 NAS-Bench-NLP . . . 141
A.5 Hardware-Aware-NAS-Bench . . . 141

B Hyperparameter **143**
B.1 Graph Neural Network-based Prediction Model . . . 143
B.2 Variational Autoencoder-based Graph Embeddings . . . 143
B.2.1 Variational Autoencoder . . . 143
B.2.2 Surrogate Model . . . 143
B.3 Generative NAS with Latent Space Optimization . . . 144
B.3.1 Generator . . . 144
B.3.2 Surrogate Model . . . 144
B.4 Is Differentiable Architecture Search truly a One-Shot Method? . . . 145
B.4.1 Computational Setup . . . 145
B.5 A Dataset for Neural Architecture Design and Robustness . . . 146

C A Dataset for Neural Architecture Design and Robustness **149**
C.1 NAS-Bench-201 . . . 149
C.2 Dataset Gathering . . . 149
C.3 Dataset Structure, Distribution, and License . . . 150
C.4 Structure . . . 151
C.5 Confidence . . . 153
C.6 Confusion Matrix . . . 157
C.7 Correlations between Image Datasets . . . 158
C.8 Example image of corruptions in CIFAR-10-C . . . 159
C.9 Main Figures for other Image Datasets . . . 159
C.9.1 CIFAR-100 Adversarial Attack Accuracies (Figure 7.2) . . . 159
C.9.2 ImageNet16-120 Adversarial Attack Accuracies (Figure 7.2) . . . 160
C.9.3 CIFAR-10-C Common Corruption Accuracies (Figure 7.4) . . . 161
C.9.4 CIFAR-100-C Common Corruption Accuracies (Figure 7.4) . . . 162
C.9.5 CIFAR-100 Adversarial Attack Correlations (Figure 7.3) . . . 163
C.9.6 ImageNet16-120 Adversarial Attack Correlations (Figure 7.3) . . . 164
C.9.7 CIFAR-100-C Common Corruption Correlations (Figure 7.5) . . . 165

D Excursus about Clustering on Graphs **167**
D.1 Auxiliary Function at Optimality . . . 167
D.2 Line by Line Description of BDCC . . . 167

D.3 Generating Feasible Integer Solutions Prior to Convergence 168

List of Algorithms **171**

List of Figures **173**

List of Tables **175**

Bibliography **177**

1 Introduction

Deep learning with Convolutional Neural Networks is the main driver for success in computer vision. This success was driven by the unprecedented results of the first image classification network (Krizhevsky et al., 2012) on the large-scale visual recognition challenge ImageNet (Deng et al., 2009). Since then, developing new architecture designs has driven research to find better and better networks for various computer vision tasks that ultimately exceed human performance (Simonyan and Zisserman, 2014; Szegedy et al., 2015; Szegedy et al., 2016; He et al., 2016). Consequently, research has shifted from feature engineering to architecture engineering. So far, architecture engineering requires human expert knowledge and experience and is characterized by many trial-and-error processes. Each network topology must be individually formed from various layer types and configuration options. The direct consequence of this is automating the architecture design process using machine learning techniques. *Neural Architecture Search* (NAS) (Elsken et al., 2019; Lindauer and Hutter, 2020; White et al., 2023) is the process that automates architecture design. It has the potential to overcome the necessary human expert knowledge and trial-and-error process and has thus gained increasing interest in recent years (Real et al., 2017; Zoph et al., 2018; Ying et al., 2019; Dong and Yang, 2020; Klyuchnikov et al., 2022; Li et al., 2021a; White et al., 2023). Well-performing architectures can be found using reinforcement learning (Zoph and Le, 2017; Zoph et al., 2018), Bayesian optimization (Kandasamy et al., 2018; Ru et al., 2021; White et al., 2021a), evolutionary algorithms (Real et al., 2017), as well as competitive baselines approaches such as random search (Li and Talwalkar, 2019) and local search (White et al., 2021b) in predefined discrete search spaces (Zoph et al., 2018; Ying et al., 2019; Dong and Yang, 2020; Liu et al., 2019; Klyuchnikov et al., 2022). The so-called cell-based search space is the most popular type of discrete search space (Zoph et al., 2018). The idea behind this type of search space is based on repeated patterns in well-performing computer vision tasks, such as ResNet (He et al., 2016). Figure 1.1 illustrates this idea based on the example of a ResNet-18 used for CIFAR-10 image classification (Krizhevsky, 2009). The architecture consists of repeating blocks (colorized in purple, blue, green, and red) that are each stacked two times to create the overall network, (Figure 1.1, top). Consequently, the convolution block, (Figure 1.1, bottom) is the repeating pattern that can be stacked several times to create a larger network, eventually leading to high performance on the downstream image task. This idea of repetitive blocks, also called cells, is taken up in the cell-based search space, firstly introduced by Zoph et al. (2018), with the goal that a NAS method searches for the optimal topology of the cells. These cells can have different topologies and operation layers (e.g., 3x3 convolution, 1x1 convolution, or pooling operations) (Ying et al., 2019; Dong and Yang, 2020; Liu et al., 2019).

However, the mentioned classical NAS methods rely on training and evaluating myriads of networks, resulting in impractical search times (Zoph et al., 2018). Recent methods focus on improving the efficiency of the search by avoiding these immense computation costs. Such methods are based on surrogate prediction models (White et al., 2021c; White et al., 2021a; Wu et al., 2021a), generative methods (Zhang et al., 2019; Yan et al., 2020) or one-shot models (Liu et al., 2019). The surrogate models embed the architecture and predict their performance. Generative models learn a latent architecture representation space and enable optimization-based search within this latent space, from which architectures are then generated. The one-shot method trains

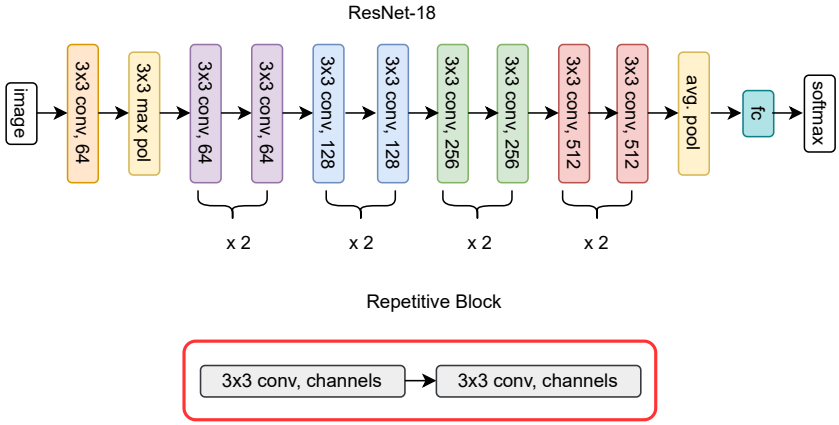


Figure 1.1: (**top**) ResNet-18 (He et al., 2016) on the CIFAR-10 image classification task (Krizhevsky, 2009). (**bottom**) Repetitive block patterns in ResNets.

and evaluates an overparameterized network from which architectures can be directly drawn with learned weights. This way, the one-shot methods can approximate the performance of an architecture. All the mentioned methods aim to improve the query efficiency of the architecture search and to reduce the computation cost. Query efficiency is crucial for NAS since each query implies full training and evaluation of the architecture on its downstream task.

The trade-off between query efficiency and finding high-performing architectures is an active research field. This thesis focuses on precisely that research field and proposes several methods to improve the efficiency of NAS search. First, we focus on presenting a surrogate model to predict the performance of architectures. The importance of using performance prediction models is furthermore studied in (White et al., 2021c). Our goal is to allow for performance prediction, especially in areas the surrogate model has not seen during training. This ability is essential for surrogate benchmarks as our NAS-Bench-301 (Zela et al., 2022), which relies on surrogate models to predict the performance of architectures in a search space of size 10^{18}. Secondly, we present two generative architecture search approaches, one based on a variational autoencoder, and the other facilitates a simple generation model. The former method enables latent space-based architecture search. In contrast, the latter improves on this model and directly optimizes the latent space for a direct generation of high-performing architectures. We further analyze the widely-used differentiable one-shot method DARTS (Liu et al., 2019) with the question whether this method is indeed efficient. This is in line with recent work (Zela et al., 2020a; Li et al., 2021b), which analyzes the stability of this one-shot method. Recently, the NAS research of finding high-performing architectures is accompanied by searching for robust architectures against adversarial attacks and corruptions. However, most work (Dong and Yang, 2019b; Devaguptapu et al., 2021; Dong et al., 2020a; Hosseini et al., 2021; Mok et al., 2021) search for architectures in the wild. In this thesis, we pave the way for comparability for this NAS research on robustness by introducing a dataset that evaluates an entire NAS search space on robustness. In total, we aim to improve the efficiency of NAS search using **topology learning for performance prediction and generative models**, as well as analyzing **one-shot methods** and paving the way for **robustness in NAS research**.

Besides the main focus of NAS topology learning for prediction, generation, and robustness, we also address discrete graph optimization using correlation clustering (Bansal et al., 2004). We

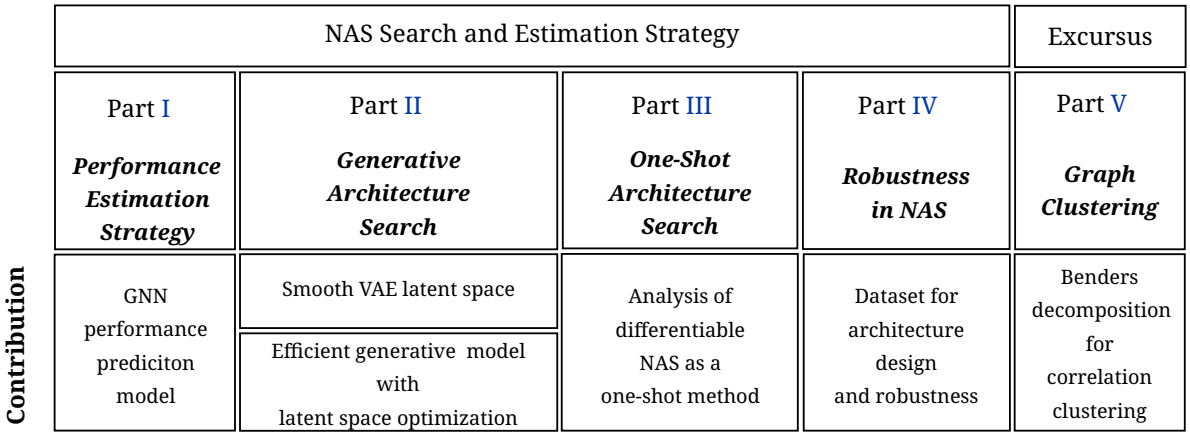


Figure 1.2: High-level thesis overview

make use of techniques from operations research (Benders, 1962; Magnanti and Wong, 1981) to improve the efficiency in correlation clustering.

Figure 1.2 gives a high-level overview of the content and structure of this thesis. In Part I, we introduce a Graph Neural Network (GNN)-based surrogate model for architecture performance prediction. This proposed method can extrapolate to unseen topological areas. In Part II, we first introduce a Graph Neural Network-based variational autoencoder (VAE), which generates a smooth architecture latent space. Smooth space in a sense, that topologically similar architectures are also close in the latent space. This smooth latent space facilitates classical NAS approaches such as Bayesian optimization and simple regression tasks to find high-performing architectures. Thereupon, we introduce a simple generative model that does not rely on an encoder model, which is coupled with a surrogate model in a fully-differentiable manner for directly generating high-performing architectures. This is possible since we additionally include a latent space optimization step, allowing for sample-efficient architecture search. Part III analyzes the widely-used differentiable architecture search in a new setting of signal recovery in terms of domain shifts and hyperparameter sensitivity. In Part IV, we introduce a dataset that evaluates all architectures in a predefined NAS search space on neural architecture robustness. Furthermore, we show first initial applications for this dataset, from robustness proxy metrics to architecture search regarding robustness. Lastly, Part V provides an excursus to discrete graph decomposition using approaches from operation research to be applied to classical correlation clustering methods for additional parallelization and speed-ups.

1.1 Contribution Overview

In this thesis, we propose several contributions to improve the efficiency of NAS by presenting surrogate models and latent space-based architecture search approaches. Lastly, we present a dataset that allows for design investigations of architectures and the influence on their corresponding robustness against adversarial attacks and corruptions.

1.1.1 *Performance Estimation Strategy*

NAS research has been made accessible to researchers without large-scale compute systems with the introduction of NAS benchmarks. The first benchmark NAS-Bench-101 (Ying et al., 2019) trained and evaluated over 423k architectures on the CIFAR-10 (Krizhevsky, 2009) image classification task. However, this benchmark comes at the cost of a restricted search space since each architecture in this predefined search space introduces an extensive evaluation. This search space restriction exposes the need for an accurate surrogate model which is able to predict the performance of architectures of structurally different and larger architectures of unseen areas, i.e., zero-shot prediction. The usage of accurate surrogate models also opened up the introduction of surrogate NAS benchmarks, with Zela et al. (2022) being the first surrogate NAS benchmark. These surrogate benchmarks furthermore allow for less restricted search space and the possibility of finding novel high-performing architectures. Recent work (White et al., 2021c) evaluates several performance prediction models for neural architecture search and analyzes the possibility for substantial prediction improvement in the case of combined prediction models. Our work is a first step in this research direction, focusing on the ability to extrapolate to unseen regions.

Contributions In Chapter 3, we introduce a surrogate model for performance prediction on the NAS-Bench-101 benchmark. This surrogate model is based on Graph Neural Networks (Zhou et al., 2020), which allows for a successful capturing of local node features and graph substructures. This capturing is essential to enable architecture mapping into a latent feature representation that captures the architecture topology. This Graph Neural Network-based surrogate model is able to accurately predict the performance of architectures in structurally different and unseen regions.

1.1.2 *Generative Architecture Search*

Classical NAS approaches use black-box methods as Bayesian optimization (Kandasamy et al., 2018; Ru et al., 2021), reinforcement learning (Zoph and Le, 2017; Zoph et al., 2018; Pham et al., 2018) or evolutionary algorithms (Real et al., 2017; Real et al., 2019) on discrete architecture search spaces. To circumvent the intrinsic discrete representation of neural architectures, works as Ying et al. (2019), White et al. (2021a), Ning et al. (2020), and Tang et al. (2020) encode neural architectures using the adjacency matrix, path encodings, or learned embeddings, which are then further coupled with black-box search methods. Another line of work uses efficient differentiable approaches (Luo et al., 2018; Liu et al., 2019), which are often biased to weight-free operations and thus often result in sub-optimal architectures (Zela et al., 2020a). The disadvantage is that the named approaches depend on the downstream task, e.g., finding an architecture with high accuracy on the CIFAR-10 image classification task. Therefore, we favor in our work the task of unsupervised learning using a variational autoencoder to generate a latent space, being independent of the downstream tasks. Figure 1.3 gives an overview of such a latent space. This latent space is learned in an unsupervised manner with the target of reconstructing the input architecture, represented by a directed acyclic graph. This latent space now allows for the architecture search strategy, facilitating black-box methods such as Bayesian optimization. The success of unsupervised learned representation for finding high-performing architecture was also shown by Zhang et al. (2019) and Yan et al. (2020). The stability and predictive power of these autoencoder-based methods rely on their

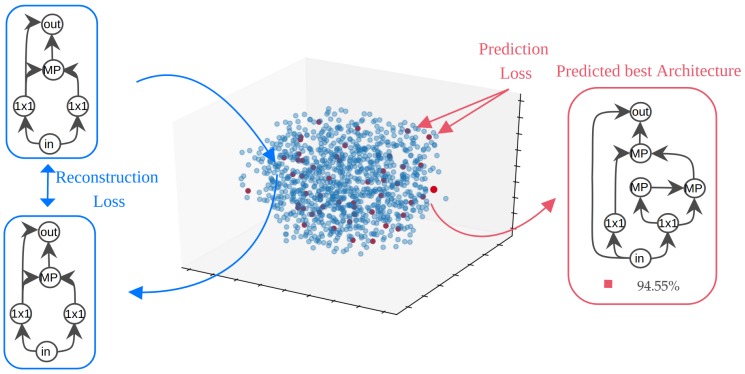


Figure 1.3: Overview of architecture latent representation space. (**left**) The process of learning a latent representation of a neural network, which can be represented as a directed acyclic graph. (**right**) A few architectures are then sampled from this learned latent space and evaluated for their ground truth information (this can be either the validation/test performance or learning curve of a downstream task, such as image classification). These architectures are then used to predict the best possible architecture in an efficient manner.

capacity to reconstruct networks correctly and to generate valid architectures from the latent space. The overall goal of learned graph representations, using a single encoder or a variational autoencoder, is to improve the search efficiency of the NAS method since each found architecture involves complete training and evaluation of the architecture on the downstream task of interest. Therefore, we present the next step, combining both search paradigms by optimizing the learned latent space. By doing that, we enable a direct generation of promising architectures.

Contributions In Chapter 4, we argue in favor of NAS on learned architecture latent spaces. We propose a two-sided graph variational autoencoders using Graph Neural Networks on the encoder and decoder level. This model builds a structurally smooth latent space by learning to reconstruct the input architecture accurately. Generating an unsupervised latent space allows for different architecture search strategy approaches, such as Bayesian optimization. We show the ability of competitive search results of our model on three different search spaces, ENAS (Pham et al., 2018), NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201 (Dong and Yang, 2020). In addition, it allows for performance prediction of architectures with topologies not observed during training.

In Chapter 5, we propose a simple generative model, which solely uses a decoder model to generate valid architectures. This generative model is paired with a surrogate model allowing for performance prediction. In addition, we incorporate a latent space optimization concept, which further allows us to iteratively reshape the latent space and thus let the model learn to generate high-performing architectures from only a few data. Instead of learning an expert prediction model, we aim to learn an expert generator for promising architectures in this work. This results in a sample-efficient search method, which achieves state-of-the-art results on several NAS benchmarks. In addition, this method allows for joint optimization in a straightforward manner.

1.1.3 *One-Shot Architecture Search*

One-shot neural architecture search was introduced to overcome the computational burden using classical NAS methods (Zoph and Le, 2017; Zoph et al., 2018; Kandasamy et al., 2018). Especially the differentiable architecture search approach, DARTS, proposed by Liu et al. (2019), is a widely researched tool. All possible architectures within a predefined search space are jointly optimized using a supernetwork in this differentiable setting. Therefore, one single optimization run is also needed to find high-performing architectures. However, this differentiable approach can lead to sub-optimal results caused by a random operation initialization, sensitivity towards the predefined search space as well as sensitivity towards training hyperparameter of the supernetwork (Zela et al., 2020a; Xu et al., 2020; Chu et al., 2020; Chen and Hsieh, 2020; Li et al., 2021b). An in-depth analysis of these mentioned causes of errors is hardly affordable in large-scale computer vision problems such as image classification, which is the primary subject area of DARTS. Therefore, we tackle in this thesis an analysis of these causes of errors in an affordable setting of inverse problems.

Contributions We analyze differentiable architecture search for inverse problems in Chapter 6. Investigating signal recovery computer vision problems instead of image classification tasks allows us to analyze the differentiable search approach w.r.t. domain shifts, training hyperparameters, network initialization, and even the impact of search space complexity in an affordable setting. Furthermore, signal recovery has received little attention in NAS, and thus we can approach this analysis without any bias to known results. This analysis allows us to investigate whether differentiable architecture search is truly a one-shot method. And indeed, differentiable NAS can find well-performing architectures in our signal recovery setting if the search space is well-preconditioned. Also, it improves over a random search baseline, especially in the case where the search space also contains not only beneficial operations but also harmful ones. The latter is crucial since it cannot always be assumed that the search space is well-preconditioned. However, this analysis also shows that the architecture reconstruction performance estimated by the supernetwork trained weights is not well correlated with the final performance after retraining the selected architecture.

1.1.4 *Robustness of NAS Architectures*

The architecture search to find novel, ever-better high-performing architectures is recently accompanied by the search for architectures being robust against adversarial attacks and corruptions. Although deep neural networks are successful on large-scale computer vision tasks, they are often highly sensitive to already small perturbations on the input data, not even visible to the human eye. This sensitivity can lead to incorrect decisions of the network with high confidence, hampering the applicability of these networks to real-world use cases. Therefore, recent NAS approaches combine the search objectives of high performance and robustness (Dong and Yang, 2019b; Devaguptapu et al., 2021; Dong et al., 2020a; Hosseini et al., 2021; Mok et al., 2021). However, these approaches use architectures in the wild, and no complete search space is used, making comparability difficult. We tackle to close this gap in our work.

Contributions Chapter 7 introduces a dataset, which evaluates an entire NAS search space, i.e., the 6 466 unique architectures from NAS-Bench-201 (Dong and Yang, 2020), in terms of robustness. This proposed dataset allows for benchmarking different NAS approaches for robustness, like classical methods such as Bayesian optimization (Snoek et al., 2015; White et al., 2021a) or random and local search (Li and Talwalkar, 2019; White et al., 2021b) or robustness proxy measurement (Mok et al., 2021). Therefore, we introduce better-streamlined research regarding architecture design for robust architectures. Having the dataset at hand, including all pretrained models with their robustness evaluations against adversarial attacks and common corruptions, we also show first applications. We evaluate all architectures in terms of common training-free measurement and their correlation to their robustness. Furthermore, we perform NAS itself and, lastly, show how the architecture design choices affect the architecture's robustness, keeping the parameter count fixed.

1.1.5 *Excursus on Graph Decomposition*

In many computer vision tasks, such as image segmentation, it is necessary to partition a set of observations into unique entities. Correlation clustering Bansal et al. (2004) is a powerful formulation on sparse graphs that tackles exactly this task. In image segmentation, the nodes of the graph correspond to superpixels, and the edges describe the adjacency between two superpixels. In addition, this formulation contains real-valued edge weights, which relate to a probability, defined by a classification model, that these two superpixels correspond to the same ground truth entity. Correlation clustering is appealing since, first, there is no need to determine the amount of segments beforehand, and second, the size of segments does not matter. However, correlation clustering is an NP-hard problem. Thus, previous methods (Andres et al., 2011; Andres et al., 2012b; Nowozin and Jegelka, 2009) are based on linear programming with cutting plane approaches, which do not scale easily to large problems. In this work, we tackle the efficiency and possibility of parallel computation.

Contributions Chapter 8 treats the correlation clustering formulation as an integer linear program and reformulates its optimization to use Benders decomposition (Benders, 1962). This is a classical approach from operations research. Furthermore, this benders decomposition approach is accelerated using Magnanti-Wong Benders rows (Magnanti and Wong, 1981), also a technique from operations research. The usage of Benders decomposition together with Magnanti-Wong Benders rows to tackle the optimization in correlation clustering theoretically allows for massive parallelization.

1.2 Outline

In the following, we specify the structure of the thesis.

Chapter 2, Related Work and Technical Background: We first provide an overview of neural architecture search, including the most commonly used search spaces and search methods in Section 2.1. Second, we provide an overview of Graph Neural Networks and their corresponding preliminaries in Section 2.2. Third, we provide technical background on Bayesian optimization as a search method and variational autoencoder in Section 2.3.

Chapter 3, Graph Neural Network-based Prediction Model: In this chapter, we propose a surrogate model with a Graph Neural Network-based encoder, which handles the different characteristics of neural architectures. This proposed surrogate model allows not only for classical supervised performance prediction but also for the performance prediction of architectures with a topology not seen during training, i.e., zero-shot prediction.
This chapter corresponds to the work Lukasik et al. (2020a), published at the DAGM GCPR 2020.

Chapter 4, Variational Autoencoder-based Graph Embeddings: This chapter introduces a two-sided graph-based variational autoencoder for neural architecture search. The proposed method is able to smoothly encode and correctly reconstruct input architectures and generate valid graphs from the latent space on various search spaces.
This chapter corresponds to the IJCNN 2021 publication Lukasik et al. (2021).

Chapter 5, Generative NAS with Latent Space Optimization: In this chapter, we further improve over the presented autoencoder from the previous chapter by introducing a simple decoder model, which learns to directly generate promising architectures facilitating an effective and sample-efficient search model.
This chapter corresponds to the ECCV 2022 publication Lukasik et al. (2020a). This work was also presented as an extended abstract at the "Third workshop on Neural Architecture Search" at CVPR 2022. This work is an equal contribution with Steffen Jung. Steffen and I developed the idea together. I implemented the generative model, the surrogate models as well as the latent space optimization for the single objective tasks, including its baselines. Steffen implemented the multi-objective search and its corresponding baselines and trained the architectures on ImageNet. We both wrote the paper jointly.

Chapter 6, Is Differentiable Architecture Search truly a One-Shot Method?: This chapter analyzes differentiable architecture search for signal recovery in terms of sensitivity towards domain shifts, hyperparameters, and network initialization. This investigation looks into the ability of differentiable architecture search to be a true one-shot method.
This chapter corresponds to Geiping et al. (2021b). Part of this work was presented at the "Workshop on Deep Learning and Inverse Problems" at NeurIPS 2021 (Geiping et al., 2021a). This work is joint work with Jonas Geiping and was jointly supervised by Margret Keuper and Michael Moeller. Jonas is an expert on signal reconstruction and built the dataset for the signal recovery experiments. I implemented the hyperparameter optimization part. We all wrote the paper jointly.

Chapter 7, A Dataset for Neural Architecture Design and Robustness: We introduce a database in which one of the most commonly considered search spaces for neural architecture search, NAS-Bench-201, is evaluated on a range of common adversarial attacks and corruption.
This chapter corresponds to Jung et al. (2023), which is accepted at ICLR 2023. This work is an equal contribution with Steffen Jung. Steffen built the dataset framework and implemented the robustness evaluations. I implemented the training-free measurement use cases and the NAS use case. We both wrote the paper jointly.

Chapter 8, Excursus about Clustering on Graphs: In this chapter, we combine two operations research formulations, i.e., Benders decomposition and Magnanti-Wong Benders rows to reformulate the optimization in classical correlation clustering to improve its efficiency and allow for massive parallelization. This chapter corresponds to Lukasik et al. (2020b), which is published in the Workshop Proceedings "Machine Learning in High Performance Computing Environments" at SC 2020.

Chapter 9, Conclusion: We conclude this thesis with a summary and present possible further research directions.

1.3 PUBLICATIONS

The following peer-reviewed papers contribute to this thesis:

- Neural Architecture Performance Prediction Using Graph Neural Networks
  Lukasik, Jovita and Friede, David and Stuckenschmidt, Heiner and Keuper, Margret
  Proc. of the German Conference on Pattern Recognition, GCPR 2020
  (Lukasik et al., 2020a)

- Smooth Variational Graph Embeddings for Efficient Neural Architecture Search
  Lukasik, Jovita and Friede, David and Zela, Arber and Hutter, Frank and Keuper, Margret
  International Joint Conference on Neural Networks, IJCNN 2021
  (Lukasik et al., 2021)

- Learning Where to Look - Generative NAS is Surprisingly Efficient
  Lukasik, Jovita* and Jung, Steffen* and Keuper, Margret
  Proc. of the European Conference on Computer Vision, ECCV 2022
  (Lukasik et al., 2022)

- DARTS for Inverse Problems: a Study on Stability
  Geiping, Jonas* and **Lukasik, Jovita*** and Keuper, Margret and Moeller, Michael
  Advances in Neural Information Processing Systems (NeurIPS), Workshop on Deep Learning and Inverse Problems., NeurIPS 2021
  (Geiping et al., 2021a)

- Neural Architecture Design and Robustness: A Dataset
  Jung, Steffen* and **Lukasik, Jovita*** and Keuper, Margret
  accepted for publication in Proc. of the International Conference on Learning Representations, ICLR 2023
  (Jung et al., 2023)

- A Benders Decomposition Approach to Correlation Clustering
  Lukasik, Jovita and Keuper, Margret and Singh, Maneesh and Yarkony, Julian
  Proc. of the IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications, MLHPC 2020
  (Lukasik et al., 2020b)

Additional publications not being part of this thesis:

- Surrogate NAS Benchmarks: Going Beyond the Limited Search Spaces of Tabular NAS Benchmarks
  Zela, Arber and Siems, Julien Niklas and Zimmer, Lucas and **Lukasik, Jovita** and Keuper, Margret and Hutter, Frank
  Proc. of the International Conference on Learning Representations, ICLR 2022
  (Zela et al., 2022)

---

* contributed equally

Related Work and Technical Background

2

In this chapter, we aim to provide a broad overview of related literature and background information as preparation for the following chapters in this thesis. We introduce *Neural Architecture Search* with its foundations and proposed approaches in Section 2.1. In Section 2.2, we describe preliminaries on *Graph Neural Networks* as the underlying framework on which many of the methods presented in this thesis rely on. Section 2.3 provides technical background on *Bayesian optimization* and *variational autoencoder*.

2.1 Neural Architecture Search

Neural architecture search (NAS) is the automative process of neural architecture design for a given task. NAS has gained substantial attention since its first improvements over human-designed neural architectures with *reinforcement learning*-based approach, proposed by Zoph and Le (2017). Since then architectures found by NAS outperform human designed architectures on different tasks and datasets. Several NAS approaches based on reinforcement learning (RL), evolutionary algorithm (EA), Bayesian optimization (BO) and weight-sharing were introduced. At the same time as new NAS methods were introduced, NAS benchmarks were also created in order to facilitate reproducible research (Li and Talwalkar, 2019; Yang et al., 2020). The foundation was laid with the first tabular benchmark in NAS, NAS-Bench-101 (Ying et al., 2019), which evaluated thousands of architectures on the CIFAR-10 (Krizhevsky, 2009) image classification task. Subsequently, many other benchmarks have been introduced on different tasks and not only image classification (Dong and Yang, 2020; Zela et al., 2022; Mehta et al., 2022; Klyuchnikov et al., 2022; Yan et al., 2021).

2.1.1 *Preliminaries*

The first NAS survey by Elsken et al. (2019) introduces NAS as a conjunction of three different parts: search space, search strategy and performance estimation strategy. Nowadays, the demarcation between search and performance estimation strategies can no longer be clearly set, as methods based on one-shot approaches directly couple the search and estimation of architectures. In the following we define NAS formally as presented in (White et al., 2023).

Definition 1 (Neural Architecture Search). *Given a search space $\mathcal{X}$, a dataset D, a training pipeline P and a predefined time budget T. In Neural Architecture Search (NAS) the goal is to find an architecture $x \in \mathcal{X}$ with the highest validation accuracy when trained on D with the training pipeline P within the time budget T by approximately solving*

$$\begin{aligned} &\min_{x \in \mathcal{X}} \mathcal{L}_{\text{val}}(w^*(x), x) \\ &\text{where } w^*(x) = \arg\min_{w} \mathcal{L}_{\text{train}}(w, x), \end{aligned} \tag{2.1}$$

with $\mathcal{L}_{\text{val}}$ and $\mathcal{L}_{\text{train}}$ being the validation and training loss on data D, w denotes the trained architecture weights.

In this thesis we use the term high-performing architectures synonymously to architectures with a high validation or test accuracy, if not stated otherwise.

2.1.2 *Search Spaces*

A search space defines the constrained set of architectures. It leads to NAS being intrinsically a discrete optimization problem seeking the optimal configuration, i.e., an architecture, in this constrained space and is the first step in the NAS procedure. The search space design and its constraints play an important role for the search method itself. Large search spaces introduce less human knowledge and bias to favored and known architecture structures as for example a ResNet-Block (He et al., 2016) and thus enable the possibility for NAS methods to find novel architectures. However, large search spaces also hamper NAS methods to find the best architecture. Yet, small search spaces introduce human knowledge and bias but allow for search speed-ups. The majority of search spaces are task-specific and indeed introduce human knowledge about state-of-the-art neural architectures. NAS-Bench-101 (Ying et al., 2019) for example covers relevant architectures such as ResNet-like (He et al., 2016) and InceptionNet-like (Szegedy et al., 2016) networks.

In the NAS research, there exists different types of search spaces (White et al., 2023), such as macro, chain-structured and hierarchical search spaces, as well as the most popular cell-based search space. In this thesis, we focus on the latter cell-based search space. For more information about the former types, we refer to the NAS surveys by Elsken et al. (2019) and White et al. (2023).

Cell-Based Search Space

Cell-based search spaces were initially introduced by Zoph et al. (2018) as a scalable and transferable approach. The motivation for this search space is based on the fact, that human designed state-of-the-art Convolutional Neural Networks (CNNS) often contain repeated patterns, as the residual block in ResNets (He et al., 2016; Szegedy et al., 2016). Therefore, Zoph et al. (2018) proposed to search for so-called cells, i.e., the repeated patterns, instead of the overall neural network. These cells are stacked several times in series to form the overall neural network, which is then trained on the task of interest, for example CIFAR-10 image classification (Krizhevsky, 2009). The outer skeleton, also called macro architecture, is predefined and the micro architecture, the cells, are searched. The former macro architecture defines the overall skeleton of how the cells are stacked, see Figure 2.1 for an example, and can also define the first stem layer, intermediate downsampling layers and the classification head.

NASNet (Zoph et al., 2018) is the first cell-based search space and consists of two types of cells in order to allow for scalable architectures: *normal cell* and *reduction cell*. The normal cell returns a feature map of the same dimension as the input feature map, whereas for the reduction cell, the initial operation has a stride of two in order to halve the height and width of the input feature map. Each cell in the NASNet search space can be represented by a directed acyclic graph (DAG). Each node in this graph is a combination of two operations, either convolution or pooling, whose outputs are combined by either an element-wise addition or a concatenation along the filter dimension. These normal and reduction cells are then stacked in an iterative manner in the predefined macro architecture, see Figure 2.1. In order to directly allow for transferability to larger datasets, the macro architecture can be varied by simply increasing the depth by stacking more normal and reduction cells in sequence.

After the release of the NASNet (Zoph et al., 2018) search space, several other cell-based search spaces were introduced (Liu et al., 2019; Ying et al., 2019; Dong and Yang, 2020). They differ in the topology of the macro architecture, as well as the cell topology and possible operations. One of the

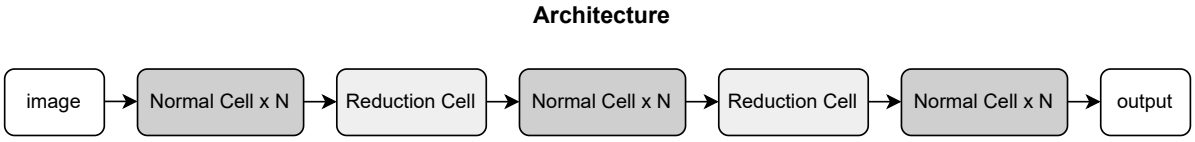


Figure 2.1: Macro architecture in the NASNet search space (Zoph et al., 2018). Gray highlighted cells differ between architectures, while the other components stay fixed. (Figure adapted from Zoph et al. (2018))

most popular search spaces is the DARTS search space (Liu et al., 2019). It also contains the search for different normal and reduction cells, but the difference to the NASNet search space is the topology of the DAG. In the DARTS search space the operation choices are on the edges and not on the nodes, where the latter represents the latent representation. In NASNet the edges represent the latent representations and the nodes define the operation choice.

Also, most of the NAS benchmarks are cell-based search spaces, from which the most popular one is NAS-Bench-101 (Ying et al., 2019). We will provide more information about NAS benchmarks as well as the DARTS search space in Section 2.1.6.

As mentioned, the search for cells instead of a complete network has the benefit that it allows for scalability and transferability to larger datasets, for example to search for cells on the CIFAR-10 image classification tasks, evaluate a larger macro architecture on ImageNet (Deng et al., 2009). In spite of the positive aspects, cell-based search spaces also face some negative aspects. These search spaces need predefined macro architectures and cell designs, which require human expert knowledge, and furthermore limits the possibility to find novel architectures. In addition, although the popular DARTS search space (Liu et al., 2019) contains 10^{18} different architecture, the variance in performance in rather small (Yang et al., 2020; Wan et al., 2022), leading to marginal improvements of NAS strategies against random search. We discuss the search spaces and its accuracy distribution in more detail in Section 2.1.6.

As previously described in the cell-based search space the architectures can be represented via DAGs, $G = (V, E)$, with V being the nodes and E being the edges in the DAG G. The operations of a neural network are either described by the nodes or edges of the DAG, depending on the topology design in the search space. With the topology of an architecture, the question is now, how to best encode the architecture's topology for the search strategies, relying on for example on mutations, similarity measurement and performance predictions. The most common architecture encoding is the adjacency matrix. White et al. (2021a) introduce a path-based encoding. Given the natural graph structure, graph-based encodings (Ning et al., 2020; Tang et al., 2020) are also utilized. Another line of work introduces learned encodings based on unsupervised learning methods as autoencoder (Zhang et al., 2019; Yan et al., 2020). Each of these encodings can be coupled with a search strategy. White et al. (2021a) evaluate the combination of different encodings and NAS search strategies and their effect on finding high-performing architectures.

These cell-based search spaces, and in general search spaces in NAS, are not restricted to vision tasks. Besides the mentioned image classification search spaces, there exists language-based search spaces, as NAS-Bench-ASR (Mehrotra et al., 2021) for speech recognition and an LSTM-based search space NAS-Bench-NLP (Klyuchnikov et al., 2022).

2.1.3 *Search Methods*

The search method is the optimization approach to find high-performing architectures in the predefined search space. Most search methods are either black-box optimization techniques or weight-sharing approaches.

The black-box approaches follow different paradigms: Reinforcement learning (RL) (Zoph and Le, 2017; Zoph et al., 2018; Pham et al., 2018) as a NAS strategy considers the neural architecture generation as the agent's action with its reward given in terms of validation accuracy. Evolutionary algorithm (EA) (Real et al., 2017; Liu et al., 2018b) approaches optimizing the neural architectures themselves by guiding the mutation of architectures and evaluating their fitness given by the validation accuracy. Early Bayesian optimization (BO) methods (Kandasamy et al., 2018) derive kernels for architecture similarity measurements to extrapolate the search space.

The initial black-box approaches were computationally heavy since they required to train and evaluate all found architectures. To overcome these computational drawbacks, faster weight-sharing approaches, also called one-shot methods, were introduced, which result in faster search approaches (Pham et al., 2018; Bender et al., 2018) and also differentiable optimization methods (Liu et al., 2019; Cai et al., 2019; Xie et al., 2019b; Zela et al., 2020a).

Other approaches map the discrete search space into a continuous architecture representation space (Luo et al., 2018; Zhang et al., 2019; Yan et al., 2020) and search or optimize within this space using for example BO (e.g., (Yan et al., 2020)) or gradient-based point operation (Luo et al., 2018). In addition to these optimization methods, Li and Talwalkar (2019) demonstrate the efficient ability of random search to find high-performing architectures and therefore point out the importance of random search being an important baseline. Also, local search (White et al., 2021b) shows the ability to find high-performing architectures in an efficient way.

In the following we will present the related work to Bayesian optimization as a search strategy.

Bayesian Optimization

Bayesian optimization (BO) is an established and powerful strategy for global optimization of expensive black-box functions (Mockus, 1974; Jones et al., 1998; Brochu et al., 2010) and has been used with significant success in NAS (Kandasamy et al., 2018; Zhang et al., 2019; White et al., 2021a). BO is based on two components: (1) building a probabilistic model to model the objective black-box function based on already observed data and (2) constructing an acquisition function, which trades off exploration and exploitation.

For the sake of clarity we will provide more technical background on BO in Section 2.3.1 and focus here on the related work for BO in NAS itself.

Initial BO approaches as Kandasamy et al. (2018) use a distance metric obtained through an optimal transport program to enable Gaussian process (GP)-based BO. White et al. (2021a) encode architectures with a high-dimensional path-based scheme and employ BO on an ensemble surrogate. Ru et al. (2021) propose a graph kernel with GP-based BO to capture the topological structure of architecture graphs. However, GP-based BO does not always perform well due to the high dimension, discrete and graph-like structure of NAS search spaces. Therefore, another line of NAS work for BO, embeds the discrete architectures into continuous encodings as described in Section 2.1.2. Shi et al. (2019) and Zhang et al. (2019) used Graph Neural Network (GNN)-based encodings to fit a Bayesian linear regression as a surrogate in BO. Note we will provide more

information about Graph Neural Networks in Section 2.2. Very recently, Yan et al. (2020) learn neural architecture representations using the proposed GNN in Xu et al. (2019) in combination with a multilayer perceptron graph decoder. The model employs a combination of the adjacency matrix and a one-hot operation encoding matrix as input for the encoder and improves over previous approaches to NAS. Their results indicate that highly informed encoding is crucial for the task.

The second component of BO is the acquisition function. However, optimizing it in each round for all possible architectures in the search space is a computationally expensive and exhaustive tasks. Therefore, most commonly the set of architectures selected as input for the acquisition function are either found by random search or local search (Ru et al., 2021; Ying et al., 2019), or the best architecture queried so far is randomly mutated (Kandasamy et al., 2018; White et al., 2021a; Ru et al., 2021) to create the input set for the acquisition function.

2.1.4 *One-Shot Methods with Supernets*

Another line of search methods are so-called *one-shot* methods. These methods were introduced to overcome the computation burden of early NAS methods relying on training and evaluating thousands of architectures (Zoph and Le, 2017; Zoph et al., 2018). Here, instead of training each architecture individually, one-shot methods implicitly train all possible architectures in a search space by training one single supernetwork. A supernetwork is an over-parameterized architecture that contains all possible architectures in the search space as subnetworks (Bender et al., 2018; Pham et al., 2018; Liu et al., 2019).

After the supernetwork is trained, each subnetwork can be directly evaluated by drawing its learned weights from the supernetwork. The underlying assumption in one-shot methods is the performance ranking similarity of the architectures using the learned supernetwork weights and the architectures retrained from scratch. There is no consistent answer, if this assumption holds true, rather some works agree with this assumption (Pham et al., 2018; Li et al., 2021b) and some show examples where this assumption does not hold true (Zela et al., 2020a). In case the ranking consistency is not met, we also speak of *rank disorder* (Li and Talwalkar, 2019; White et al., 2023)

Also in one-shot methods, the question is how to best search for architectures. There exists two approaches: supernet training in conjunction with a black-box optimization techniques (Bender et al., 2018; Pham et al., 2018) or differentiable optimization approaches (Liu et al., 2019).

Supernet methods with Black-Box Optimization

Some methods decouple the supernet training from the search process (Bender et al., 2018; Li and Talwalkar, 2019; Guo et al., 2020b) whereas others other work combines the supernet training with the search approach (Pham et al., 2018). For the former, one architecture (subnetwork) is randomly sampled from the search space in each training step and only the weights for this architecture are updated in the supernetwork. After the supernetwork is trained, architectures can be randomly sampled (Bender et al., 2018; Li and Talwalkar, 2019) or searched by evolutionary approaches (Guo et al., 2020b) and evaluated based on their one-shot performance. The best found architecture is then trained from scratch.

In contrast to decoupled approaches, Pham et al. (2018) propose *ENAS*, a two stage approach based on the reinforcement learning from Zoph and Le (2017). First, an architecture is selected,

trained and then the weights of the supernetwork are updated. Second, the weights of the controller are updated by sampling several architectures from the supernetwork with their corresponding one-shot performance, where the reward for the controller is computed on the validation set of the downstream task, e.g., CIFAR-10 image classification. We will provide more detailed information about the ENAS search space in Section 4.4.

Differentiable Supernet Methods

Differentiable neural architecture search was first introduced by Liu et al. (2019) as *DARTS*. This approach is based on a continuous relaxation of the discrete architectures in order to allow for gradient descent to find high-performing architectures (subnetworks) from the search space spanned by the supernet. This method optimizes cells and is therefore applicable to any cell-based search space, which is defined with operations on the edges. Recall here Section 2.1.2.

For the DARTS search space each node $x^{(i)}$ in the DAG corresponds to the latent representation and each directed edge (i, j) consists of a set of multiple candidate operations $o^{(i,j)}$ which transform the node $x^{(i)}$. The candidate operations on the edge (i, j) are parameterized by *architecture parameters* $\alpha_o^{(i,j)}$. Therefore during the supernet training, each operation on the edge (i, j) is weighted by its architecture parameter $\alpha_o^{(i,j)}$. The search strategy in DARTS is formulated as a bilevel optimization problem, which jointly optimizes the architecture parameters $\alpha = (\alpha_o^{(i,j)})$ and the network weight parameters w. Note, the network weight parameters also include the weights of all candidate operations. The bilevel optimization is approximated by a single step of gradient descent to overcome the expensive inner optimization. After training the supernet, the discrete architecture is obtained by selecting the operations $o^{(i,j)}$ for each edge as the operation with the highest architecture parameter $\alpha_o^{(i,j)}$. This architecture is then trained from scratch.

DARTS shows significant advantages in the search of high-performing neural architectures within only a few GPU days. Building on this pioneering work NAS research has gained significant momentum for further improvements over the original DARTS approach (Dong and Yang, 2019b; Cai et al., 2019; Xie et al., 2019b; Chen et al., 2019; Akimoto et al., 2019; Xu et al., 2020; He et al., 2020; Chen and Hsieh, 2020; Wu et al., 2021b; Zhang et al., 2021). For example Chen et al. (2019) present an algorithm to progressively increase the depth of the searched architecture during training, bridging the gap between search and evaluation performances.

Stability of DARTS There are only few works investigating the stability of DARTS in terms of rank disorder or poor test generalization after retraining (Zela et al., 2020a; Xu et al., 2020; Chu et al., 2020; Chen and Hsieh, 2020; Li et al., 2021b). RobustDARTS (Zela et al., 2020a) tracks the dominant eigenvalue $\lambda_{\max}^{\alpha}$ of the Hessian during the architecture search and implement a regularization and early stopping criterion based on this quantity for a more robust DARTS search. Chen and Hsieh (2020) pick up the relationship between the Hessian during the architecture search and the performance gap during search and evaluation time. They propose a perturbation-based regularization to smooth the validation loss landscape. Xu et al. (2020) find that only connecting partial channels into the operation selection leads to a regularized search to improve the stability. Chu et al. (2020) use a sigmoid activation for the architecture weights instead of softmax to eliminate unfair optimization regarding the skip-connection operation. Yang et al. (2020) analyze the contribution of each component in a NAS approach within the search space from Liu et al. (2019). They highlight that a performance-boosting training pipeline, often a result of expert

knowledge, is more important for the evaluation of architectures than the search itself. Li et al. (2021b) use single-level optimization to improve the rank disorder and poor test generalization. Furthermore, the architecture parameters are re-parameterized over the simplex and updated using exponentiated gradient, leading to an approach not relying on retraining.

We further investigate this differentiable supernet method and its stability in Chapter 6.

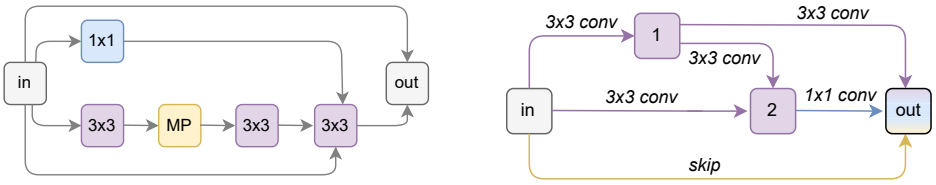
2.1.5 *Performance Estimation Strategy*

All NAS approaches are dependent on *performance estimation* of intermediate architectures. To avoid the computation-heavy training and evaluation of queries on the target dataset, methods to approximate the performance have been explored (White et al., 2021c). The motivation behind using performance estimation models for the architecture search process is given by the fact that evaluating an estimation model on an architecture in a given search space, which estimation the accuracy of this architecture, takes less time than fully training this architecture. Ideally, the correlation of the estimation accuracy is high with the true accuracy. This estimation is done by using performance prediction models. Predicting the performance of neural networks based on features such as the network architecture, training hyperparameters or learning curves has been exploited previously via MCMC methods (Domhan et al., 2015), Bayesian Neural Networks (Klein et al., 2017) or regression models (Baker et al., 2017). (Long et al., 2020) manually construct features to regress neural networks or support vector regressors. Liu et al. (2018a) use a performance predictor in an iterative manner during the search process of NAS. Baker et al. (2018) use features of a neural architecture, such as the validation accuracy, additionally architecture parameters such as the number of weights and the number of layers, as well as hyperparameters, to predict learning curves during the training process by means of a sequential regression models. Luo et al. (2018) propose a performance prediction model learned in combination with an auto-encoder in an end-to-end manner. The neural architectures are mapped into a latent feature representations, which are then used by the predictor for performance prediction and are then further decoded into neural architectures. Common approaches include neural predictors that take path encodings (White et al., 2021a) or graph encodings learned by Graph Neural Networks (Shi et al., 2019; Wen et al., 2020; Ning et al., 2020; Tang et al., 2020) as input. Recently, WeakNAS (Wu et al., 2021a) proposes to progressively evaluate the search space towards finding high-performing architectures using a set of weak predictors.

2.1.6 *Benchmarks*

NAS, as defined in Definition 1, aims to find an architecture that is high-performing e.g., on image classification tasks like CIFAR-10 (Krizhevsky, 2009). As a consequence, different search spaces were introduces, with different training pipelines and computation costs. This however made it difficult to compare different NAS methods. The first NAS benchmark, NAS-Bench-101 (Ying et al., 2019), led to a paradigm shift in the comparability and statistical significance of NAS methods and further helped to reduce the computation time of developing and evaluating NAS methods.

Definition 2 (NAS Benchmarks). *Following Lindauer and Hutter (2020) a NAS benchmark is defined by containing a dataset with a predefined training-test split, a search space and a fixed training pipeline for training the architectures on the dataset (including runnable code).*



(a) NAS-Bench-101 cell.

(b) NAS-Bench-201 cell.

Figure 2.2: Visualization of best cells in the tabular benchmarks in terms of mean test accuracy on CIFAR-10. **(left)** Best cell in NAS-Bench-101 with a test accuracy of 94.32%. **(right)** Best cell in NAS-Bench-201 with a test accuracy of 94.37%

Build on that, a tabular NAS benchmark also gives precomputed metrics (as training and validation/test performance) for all architectures in the search space of the NAS benchmark.

A surrogate NAS benchmark evaluates a portion of a usually very large search space for precomputed performance metrics and furthermore includes a surrogate model which predicts the performance of any architecture in the search space.

The first tabular NAS benchmark is *NAS-Bench-101* (Ying et al., 2019). The search space is a cell-based search space and contains 423 624 unique neural networks. Each architecture is trained on CIFAR-10 (Krizhevsky, 2009) for image classification. The cell topology is limited to the number of nodes $|V| \leq 7$ (including input and output nodes) and edges $|E| \leq 9$. The nodes represent the network layers and intermediate nodes can take any operation from the operation set $O = \{1 \times 1 \text{ conv.}, 3 \times 3 \text{ conv.}, 3 \times 3 \text{ max pooling}\}$. While this search space is limited it covers relevant architectures such as for example ResNet like (He et al., 2016) and InceptionNet like (Szegedy et al., 2016) models (Ying et al., 2019). Figure 2.2 (left) visualizes the best cell in NAS-Bench-101 in terms of mean test accuracy over 3 runs. Zela et al. (2020b) show, that only subspaces of the architectures in NAS-Bench-101 can be used to evaluate one-shot NAS methods (Liu et al., 2019; Pham et al., 2018), motivating their proposed variant NAS-Bench-1shot1 (Zela et al., 2020b).

Similarly to NAS-Bench-101, *NAS-Bench-201* (Dong and Yang, 2020) uses a restricted, cell-structured search space, while the employed graph representation allows evaluating discrete and one-shot NAS algorithms. NAS-Bench-201 consists of 15 625 architectures, from which in total 6 466 architectures are unique. Each architecture is trained for 200 training epochs on CIFAR-10 (Krizhevsky, 2009), CIFAR-100 (Krizhevsky, 2009), and ImageNet16-120 (Chrabaszcz et al., 2017) image classification tasks. This benchmark provides validation and test accuracy information for each of the three datasets. The cell structure is different compared to NAS-Bench-101: Each cell has $|V| = 4$ nodes and $|E| = 6$ edges, where the former represent feature maps and the latter denote operations chosen from the set $O = \{1 \times 1 \text{ conv.}, 3 \times 3 \text{ conv.}, 3 \times 3 \text{ avg pooling}, \text{skip}, \text{zero}\}$. Figure 2.2 (right) visualizes the best cell in NAS-Bench-201 in terms of mean test accuracy on CIFAR-10.

In addition to these tabular benchmarks *NAS-Bench-301* (Zela et al., 2022) (now called Surr-NAS-Bench-DARTS) provides the first surrogate benchmark, which allows for fast evaluation of NAS methods on the DARTS (Liu et al., 2019) search space by querying the validation accuracy. NAS-Bench-301 evaluates several surrogate models on in total 60 000 sampled architectures from the DARTS (Liu et al., 2019) search space on the CIFAR-10 (Krizhevsky, 2009) image classification

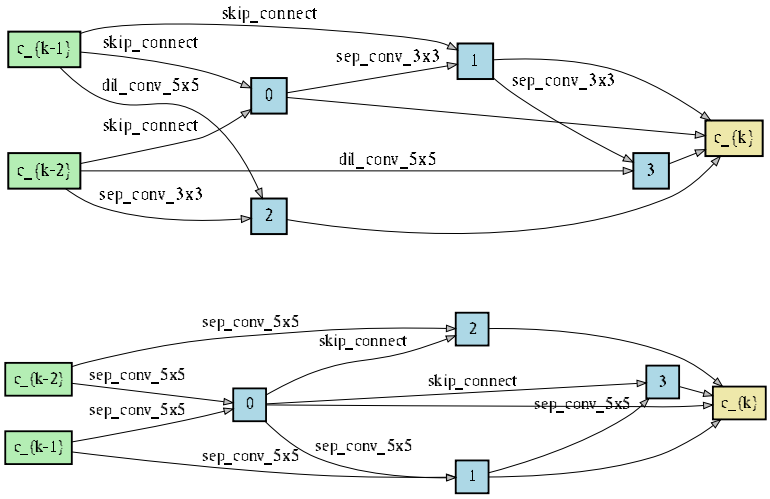


Figure 2.3: Visualization of the normal and reduction cell of the best architecture in the surrogate benchmark NAS-Bench-301 in terms of validation accuracy on CIFAR-10 94.73%. (**top**) Normal cell. (**bottom**) Reduction cell.

task. The DARTS search space consists of 10^{18} neural networks, where each network consists of two cells; a normal cell and a reduction cell. Each cell is limited by the number of nodes $|N| = 7$ and the number of edges $|E| = 12$, where 4 of these edges connect the intermediate nodes (excluding the input nodes) to the output node. Each edge denotes an operation from the set $O = \{3 \times 3$ sep. conv., 5×5 sep. conv., 3×3 dil. conv., 5×5 dil. conv., 3×3 avg pooling, 3×3 max pooling, identity, zero$\}$. Each intermediate edge is connected to two predecessor nodes. Each cell also contains two input nodes, which are the output nodes from the previous two cells. The overall network is created by stacking the normal and reduction cell. Figure 2.3 visualizes the normal and reduction cell of the best architecture in NAS-Bench-301 in terms of test accuracy on CIFAR-10.

In addition to NAS-Bench-301, Zela et al. (2022) also released Surr-NAS-Bench-FBNet evaluated on the FBNet search space (Wu et al., 2019). Following this surrogate benchmarks, NAS-Bench-x11 (Yan et al., 2021) allows for learning curve predictions, by containing full training information at each epoch for each architecture in their considered search spaces. TransNAS-Bench-101 (Duan et al., 2021) introduces a benchmark containing performance and metric information across different vision tasks.

NAS-Bench-NLP (Klyuchnikov et al., 2022) is the first RNN-derived benchmark for language modeling tasks. From the total 10^{53} possible architectures in the complete search space, 14 322 architectures are trained on Penn TreeBank (PTB) (Mikolov et al., 2010) and provided in this benchmark. The cell search space is constrained by the number of nodes with at most 24, the number of hidden states less than 3 and the number of linear input vectors to maximally 3. The nodes represent the architecture operational layer and are chosen from the set O = {linear, element wise blending, element wise product, element wise sum, Tanh activation, Sigmoid activation, LeakyReLU activation}.

Figure 2.4: Random examples from the CIFAR-10 image classification dataset (Krizhevsky, 2009) for each of the 10 classes.

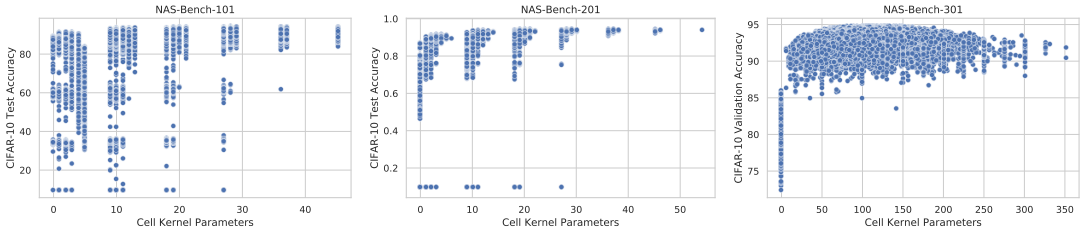


Figure 2.5: (**left**) Test accuracy (in %) on CIFAR-10 by kernel parameters ($[0, 45]$) for all architectures in NAS-Bench-101. (**middle**) Test accuracy (in %) on CIFAR-10 by kernel parameters ($[0, 54]$) for all architectures in NAS-Bench-201. (**right**) Validation accuracy (in %) on CIFAR-10 by kernel parameters ($[0, 350]$) for all sampled architectures in NAS-Bench-301 from the DARTS search space.

The recently introduced *HW-NAS-Bench* (Li et al., 2021a) is the first public dataset for hardware NAS. It extends two representative NAS search spaces, NAS-Bench-201 (Dong and Yang, 2020) and FBNet (Wu et al., 2019), by providing measured and estimated hardware costs (i.e., latency and/or energy) for each device for all architectures in both search spaces. For this, HW-NAS-Bench considers six hardware devices: *Edge GPU* (*NVIDIA Jetson TX2*), *Raspi 4* (*Raspberry Pi Limited*), *Edge TPU* (*Google LLC. Edge TPU Compiler*), *Pixel 3* (*Google LLC. Pixel 3*), *ASIC-Eyeriss* (Chen et al., 2017) and *FPGA* (*Xilinx Inc. Vivado High-Level Synthesis* ; *Xilinx zynq-7000 soc zc706 evaluation kit*). NAS-Bench-360 (Tu et al., 2022) is a benchmark suite for 10 different tasks, from which 3 use the pretrained architectures provided by NAS-Bench-201. Lastly, NAS-Bench-Suite (Mehta et al., 2022) combines 28 NAS benchmarks into one interface, allowing for reproducible search on all these benchmarks.

The image classification-based NAS benchmarks, NAS-Bench-101 (Ying et al., 2019), NAS-Bench-201 (Dong and Yang, 2020) and NAS-Bench-301 (Zela et al., 2022), evaluate all architectures on the CIFAR-10 image dataset (Krizhevsky, 2009). This dataset contains images with low resolution, as can be seen in Figure 2.4, which enables a rather fast training of architectures. However, we are mostly interested in finding novel high-performing architectures on larger more complex datasets as ImageNet (Deng et al., 2009). As mentioned in (Zoph et al., 2018; White et al., 2023) for these larger datasets, the high-performing cells on smaller dataset as CIFAR-10 are mostly used and are stacked to a larger network for ImageNet training and evaluation. A search approach on this larger datasets is also recently done by proxy metrics. These are performance estimation techniques, which assign scores to each architecture based on fast computations such as one forward and backward pass on a single minibatch. These scores are hoped to be correlated with the final accuracy of the architecture (Mellor et al., 2021).

Finding high-performing architectures on CIFAR-10 in the different NAS benchmark search spaces is also a rather simple tasks, since the variance of the architectures performances is rather small. We visualize in Figure 2.5 the test accuracy on CIFAR-10 against the architectures' cell kernel parameters. We count 1 for each 1×1 convolution, 9 for each 3×3 convolution in the cell and additionally 6 for each 3×3 separable convolution and 10 for each 5×5 separable convolution in NAS-Bench-301. Overall, we can see for NAS-Bench-201 and NAS-Bench-301 the variance is indeed rather small and a large portion of the architectures have a high test performance on CIFAR-10. NAS-Bench-101 is more spread, but also here we can see that the biggest portion of architectures in this search space have a high test accuracy. Note, the goal for NAS methods in these search spaces (NAS-Bench-101, NAS-Bench-201) is not to find only any high-performing architecture but the rather difficult task, namely finding the global optimum.

2.2 Graph Neural Networks

Combining modern machine learning methods with graph structured data has increasingly gained popularity. One can interpret it as an extension of deep learning techniques to non-Euclidean data (Bronstein et al., 2017) or even as inducing relational biases within deep learning architectures to enable combinatorial generalization (Battaglia et al., 2018). Because of the discrete nature of graphs, they can not trivially be optimized in differentiable learning methods that act on continuous spaces. The concept of Graph Neural Networks (GNNs) is a remedy to this limitation. The idea of GNNs as an iterative process which propagates the node states until an equilibrium is reached, was initially mentioned in 2005 by Gori et al. (2005). Motivated by the increasing popularity of CNNs, Bruna et al. (2014) and Henaff et al. (2015) defined graph convolutions in the Fourier domain by utilizing the graph Laplacian. The modern interpretation of GNNs was first mentioned in Li et al. (2016), Niepert et al. (2016), and Kipf and Welling (2017) where node information was inductively updated through aggregating information of each node's neighborhood. This approach was further specified and generalized by Hamilton et al. (2017) and Gilmer et al. (2017).

The research in GNNs enabled breakthroughs in multiple areas related to graph analysis such as computer vision (Xu et al., 2017; Landrieu and Simonovsky, 2018; Yi et al., 2017), natural language processing (Bastings et al., 2017), recommender systems (Monti et al., 2017), chemistry (Gilmer et al., 2017) and others.

2.2.1 *Preliminaries*

In the following, we introduce the notation of GNN models. Note, since we consider neural networks as graphs, these graphs are directed and acyclic. In the course of the thesis, we retain this assumption unless stated otherwise. Let $G = (V, E)$ be a directed acyclic graph with nodes $v \in V$ and edges $e \in E \subset V \times V$. GNNs are used to learn node representation vectors $\mathbf{h}_v$ for each node $v \in V$, and based on these node representations, we can also use GNNs to learn a graph representation vector $\mathbf{h}_G$.

Standard GNNs can be seen as a two-step procedure. In the first step the GNN learns a representation for each node $v \in V$, by iteratively aggregating the representations of neighboring nodes using an aggregation function $\mathcal{A}(\cdot)$. Then it updates the representation with the update

function $\mathcal{U}(\cdot)$. After K rounds of iterations, the final representation of each node v is computed. The second step computes a graph representation $\mathbf{h}_G$ by aggregating the node representations.

For the next detailed information, we follow the notation in Xu et al. (2019).

Node Representation

The first step in a GNN is the node information propagation. For that, we first denote $\mathcal{V}(v) = \{u \in V \mid (u,v) \in E\}$ as a set of adjacent nodes to v. For each node $v \in V$, we associate an initial node embedding $\mathbf{h}_v^{(0)} \in \mathbb{R}^{d_n}$. GNNs iteratively aggregate node representations of a node's neighbor, followed by a representation update for k iterations. The k iterations of these aggregation-update steps capture the k-hop neighborhood information for each node v. The aggregation and update step, sometimes also called *message passing* process, can be written as (Xu et al., 2019):

$$\mathbf{a}_v^{(k)} = \mathcal{A}^{(k)}\left(\left\{\mathbf{h}_u^{(k-1)} : u \in \mathcal{V}(v)\right\}\right), \tag{2.2}$$

$$\mathbf{h}_v^{(k)} = \mathcal{U}^{(k)}\left(\mathbf{h}_v^{(k-1)}, \mathbf{a}_v^{(k)}\right), \tag{2.3}$$

with $\mathcal{A}(\cdot)$ being differentiable, permutation invariant aggregation function, $\mathcal{U}(\cdot)$ being a differentiable update module and $\mathbf{h}_v^{(k)}$ the node representation of node v after k iteration steps.

Graph Representation

After the final round of message passing, the propagated node representations $\left(\mathbf{h}_v^{(K)}\right)_{v \in V}$ are used to compute the overall graph representation:

$$\mathbf{h}_G = \text{READOUT}\left(\left\{\mathbf{h}_v^{(K)} \middle| v \in V\right\}\right), \tag{2.4}$$

$\mathbf{h}_G \in \mathbb{R}^{d_g}$ and where READOUT can be a simple summation as in Xu et al. (2019).

2.3 TECHNICAL BACKGROUND

This section provides technical background on Bayesian optimization and variational autoencoder.

2.3.1 *Bayesian Optimization*

Given a function $f : \mathcal{X} \to \mathbb{R}, \mathcal{X} \subset \mathbb{R}^d$ a compact set, we can define Bayesian optimization (BO) as an optimization problem to globally solve:

$$\max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}). \tag{2.5}$$

The function f is often assumed to be a black-box function, with no known structure and no simple closed form. Typically f is expensive to evaluate and outputs noisy evaluations $y \in \mathbb{R}, y = f(\mathbf{x}) + \varepsilon, \varepsilon \sim \mathcal{N}(0, \sigma^2)$, with expected objective function value $\mathbb{E}[y|f(\mathbf{x})] = f(\mathbf{x})$ (Shahriari et al., 2016).

In the setting of NAS, the optimization problem aims to find an architecture

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

Algorithm 1: Bayesian Optimization

Input: (i) data $D_{N-1} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N-1}$
Input: (ii) objective function f
Input: (iii) probabilistic distribution function $\mathcal{P}$
Input: (iv) acquisition function a
Input: (v) data query budget T
for $N \leq T$ **do**
  fit posterior distribution $\mathcal{P}$ on data D_{N-1}
  use acquisition function to sample $\mathbf{x}_N = \arg\max_{\mathbf{x}\in\mathcal{X}} a_{\mathcal{P}(f|D_{N-1})}(x)$
  evaluate $y_N = f(\mathbf{x}_N) + \varepsilon$
  augment data $D_N \leftarrow D_{N-1} \cup (\mathbf{x}_N, y_N)$
  $N \leftarrow N + 1$
end

where $\mathbf{x}^*$ is the global optimal architecture in $\mathcal{X}$ and $f(\mathbf{x})$ is the validation performance of $\mathbf{x}$ evaluated on a task of interest with data D, e.g., image classification on CIFAR-10 (Krizhevsky, 2009), see Definition 1.

BO models the black-box function f via a probabilistic surrogate model and exploits this surrogate model to find the next location in $\mathcal{X}$ for evaluation by means of the objective function f.

The prior distribution $\mathcal{P}(f)$ captures prior beliefs about the function f. Given observed data $D_N = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$ and the likelihood $\mathcal{P}(D_N|f)$, the posterior distribution $\mathcal{P}(f|D_N)$ can be calculated based on Bayes' theorem for tractable probabilistic models. This posterior distribution represents updated beliefs about f.

This posterior distribution is now used to find the next location in $\mathcal{X}$ to be sampled. This is done by constructing an acquisition function $a_{\mathcal{P}(f|D_N)} : \mathcal{X} \to \mathbb{R}$ based on the current posterior. The acquisition function trades off exploration and exploitation, where the former introduces locations, in which the surrogate function is uncertain, and the latter focuses on locations with high expected objective function values.

Having now both components at hand, the overall Bayesian optimization process is an iterative process by updating the posterior distribution with new datapoints proposed by the acquisition function. The iterative process is described in Algorithm 1 (Brochu et al., 2010).

Probabilistic Model

As discussed in the previous part, BO constructs a probabilistic model $\mathcal{P}(f)$ of the objective function f. In the following we will give more details about Gaussian processes and its sparse variant as probabilistic models.

Gaussian Processes The most common choice for the prior probabilistic model $\mathcal{P}(f)$ are Gaussian processes (GP) (Rasmussen and Williams, 2006; Osborne et al., 2008). A GP is a collection of random variables, such that any finite number of random variables follows a multivariate Gaussian distribution (Rasmussen and Williams, 2006). A GP is completely defined by its mean $m\colon \mathcal{X} \to \mathbb{R}$ and covariance function (also known as kernel) between two points $K(\mathbf{x}, \mathbf{x}')\colon \mathcal{X}\times\mathcal{X} \to \mathbb{R}$, and may be denoted by $GP_{m,K} = (GP_{m,K}(\mathbf{x}))_{\mathbf{x}\in\mathcal{X}}$. For fixed $\mathbf{x} \in \mathcal{X}$, $GP_{m,K}(\mathbf{x})$ returns a normally distributed

random variable with mean $m(\mathbf{x})$ and variance $K(\mathbf{x},\mathbf{x})$ and for multiple $\mathbf{x}_1,\dots,\mathbf{x}_N \in \mathcal{X}$, the vector $(GP_{m,K}(\mathbf{x}_i))_{i=1,\dots,N}$ is multivariate Gaussian with distribution $\mathcal{N}((m(\mathbf{x}_i)_{i=1,\dots,N}, (K(\mathbf{x_i},\mathbf{x}_j))_{i,j=1,\dots,N})$. The dependence structure at different locations is therefore captured by the covariance kernel K. The mean $m(\mathbf{x})$ is often set to 0 for all $\mathbf{x} \in \mathcal{X}$, in which case the GP is said to be centered.

A popular choice for the covariance function $K(\mathbf{x},\mathbf{x}')$ is the Radial Basis Function (RBF):

$$K_\theta(\mathbf{x},\mathbf{x}') = \nu \exp\Big(-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2l}\Big), \quad \theta = (\nu, l). \tag{2.6}$$

This covariance functions approaches ν for close inputs, and quickly converges to 0 as the distance increases, thus expressing weak dependence between inputs at sufficiently distinct locations. This covariance function depends on internal kernel hyperparameter $\nu > 0, l > 0$, where l changes the length-scale and ν controls the overall variance of the process. For the sake of presentation, we fix this covariance kernel in the following.

Given data $D_N = \{(\mathbf{x}_n, y_n)\}_{n=1}^N = (\mathbf{X},\mathbf{y})$, with $y_n = f(\mathbf{x}_n) + \varepsilon_n$ with $(\varepsilon_n) \sim \mathcal{N}(0,\sigma^2\mathbb{1})$ independent of $\bar{\mathbf{f}} := (f(\mathbf{x}_n))_{n=1,\dots,N}$, we aim to model and update the prior model for the function evaluation $\mathbf{f}^* = f(\mathbf{x}^*)$ for some new data point $\mathbf{x}^* \in \mathcal{X}$ by means of *Bayes' rule*.

The following is based on Snelson and Ghahramani (2005) and Rasmussen and Williams (2006). Let $\mathbf{K}_N^\theta = K_\theta(\mathbf{X},\mathbf{X}) := (K_\theta(\mathbf{x}_i,\mathbf{x}_j))_{i,j=1,\dots,N}$, $K_{\mathbf{X}*}^\theta := K_\theta(\mathbf{X},\mathbf{x}^*) \in \mathbb{R}^{N\times 1}$ and $K_{**}^\theta := K_\theta(\mathbf{x}^*,\mathbf{x}^*)$. We assume the prior on $(\bar{\mathbf{f}},\mathbf{f}^*)$ to be

$$\mathcal{P}_{\theta,\sigma}((\bar{\mathbf{f}},\mathbf{f}^*)|\mathbf{X},\mathbf{x}^*) = \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K}_N^\theta & K_{\mathbf{X}*}^\theta \\ (K_{\mathbf{X}*}^\theta)^\top & K_{**}^\theta \end{bmatrix}\right).$$

Let the noisy observation y^* at location $\mathbf{x}^*$ be given by $y^* = f(\mathbf{x}^*)+\varepsilon$ with $\varepsilon \sim \mathcal{N}(0,\sigma^2)$ independent of (ε_n). By the noise assumptions the conditional distribution of $(\mathbf{y},y^*)$ given the noise-free function evaluations $(\bar{\mathbf{f}},\mathbf{f}^*)$ is Gaussian $\mathcal{P}((\mathbf{y},y^*)|\bar{\mathbf{f}},\mathbf{f}^*) = \mathcal{N}((\bar{\mathbf{f}},\mathbf{f}^*),\sigma^2\mathbb{1})$, and the marginal likelihood of the noisy observations $\mathbf{y}$ is given by

$$p_{\theta,\sigma}(\mathbf{y}|\mathbf{X}) = \mathcal{N}(\mathbf{y}|\mathbf{0},\mathbf{K}_N^\theta + \sigma^2\mathbb{1}). \tag{2.7}$$

The marginal likelihood as a function of the hyperparameters (θ,σ) can be used as a maximization objective to fit the hyperparameters to the data using gradient descent (Snelson and Ghahramani, 2005).

Having determined (locally) optimal values $(\theta_\star,\sigma_\star)$ for these hyperparameters, we can make predictions for $\mathbf{f}^*$ by using a closed form expression of the posterior based on *Bayes' rule* (Rasmussen and Williams, 2006, Eq. 2.22-2.24):

$$\mathcal{P}_{\theta_\star,\sigma_\star}(\mathbf{f}^*|\mathbf{x}^*,\mathbf{X},\mathbf{y}) = \mathcal{N}((K_{\mathbf{X}*}^{\theta_\star})^\top(\mathbf{K}_N^{\theta_\star}+\sigma_\star^2\mathbb{1})^{-1}\mathbf{y}, K_{**}^{\theta_\star} - (K_{\mathbf{X}*}^{\theta_\star})^\top((\mathbf{K}_N^{\theta_\star}+\sigma_\star^2\mathbb{1})^{-1}K_{\mathbf{X}*}^{\theta_\star}). \tag{2.8}$$

The Gaussian predictive distribution for $\mathbf{x}^*$ is accordingly given by

$$\mathcal{P}_{\theta_\star,\sigma_\star}(y^*|\mathbf{x}^*,\mathbf{X},\mathbf{y}) = \mathcal{N}((K_{\mathbf{X}*}^{\theta_\star})^\top(\mathbf{K}_N^{\theta_\star}+\sigma_\star^2\mathbb{1})^{-1}\mathbf{y},\ \sigma_\star^2 + K_{**}^{\theta_\star} - (K_{\mathbf{X}*}^{\theta_\star})^\top(\mathbf{K}_N^{\theta_\star}+\sigma_\star^2\mathbb{1})^{-1}K_{\mathbf{X}*}^{\theta_\star}). \tag{2.9}$$

However, the inversion of the covariance matrix leads to training time of a Gaussian process of order $O(N^3)$ for N training data (Snelson and Ghahramani, 2005), which is prohibitively large for many applications. This motivates sparse Gaussian processes, which we introduce next.

Sparse Gaussian Process Given the cubic training time, Snelson and Ghahramani (2005) introduce *sparse Gaussian processes* (SGP), which introduce a small noise-free pseudo data set $\widetilde{D}_M = \{(\widetilde{\mathbf{x}}_m, f(\widetilde{\mathbf{x}}_m))\}_{m=1}^M = (\widetilde{\mathbf{X}}, \widetilde{\mathbf{f}})$, with $M \ll N$. For now let us assume the pseudo inputs to be similarly distributed as the real data D_N and accordingly place the same prior on the pseudo targets $\widetilde{\mathbf{f}}$, that is

$$\mathcal{P}_{\theta,\sigma}(\widetilde{\mathbf{f}}|\widetilde{\mathbf{X}}) = \mathcal{N}(\mathbf{0}, \mathbf{K}_M^\theta), \tag{2.10}$$

with $\mathbf{K}_M^\theta = K_\theta(\widetilde{\mathbf{X}}, \widetilde{\mathbf{X}})$. Likewise, the prior on $(\mathbf{f}, \widetilde{\mathbf{f}})$ is given as the distribution of the Gaussian process GP_{0,K_θ} evaluated at locations $(\mathbf{X}, \widetilde{\mathbf{X}})$. The corresponding likelihood for the target data $\mathbf{y}$ conditioned on $\mathbf{X}$ and the complete pseudo data $(\widetilde{\mathbf{X}}, \widetilde{\mathbf{f}})$ then becomes

$$p_{\theta,\sigma}(\mathbf{y}|\mathbf{X}, \widetilde{\mathbf{X}}, \widetilde{\mathbf{f}}) = \mathcal{N}(\mathbf{y}|\mathbf{K}_{NM}^\theta (\mathbf{K}_M^\theta)^{-1}\widetilde{\mathbf{f}}, \mathbf{\Lambda} + \sigma^2 \mathbb{1}), \tag{2.11}$$

with $\mathbf{K}_{NM}^\theta = K_\theta(\mathbf{X}, \widetilde{\mathbf{X}})$, $\mathbf{\Lambda} = \mathrm{diag}((\lambda_n)_{n=1,\dots,N})$, where

$$\lambda_n = K_\theta(\mathbf{x}_n, \mathbf{x}_n) - (\mathbf{k}_n^\theta)^\top (\mathbf{K}_M^\theta)^{-1}\mathbf{k}_n^\theta, \quad (\mathbf{k}_n^\theta)^\top = (K_\theta(\widetilde{\mathbf{x}}_m, \mathbf{x}_n))_{m=1,\dots,M}.$$

The next step now is to find the posterior distribution given the prior (Equation (2.10)) and the likelihood (Equation (2.11)). According to Bayes' rule the pseudo target posterior is given by

$$\mathcal{P}_{\theta,\sigma}(\widetilde{\mathbf{f}}|\mathbf{X}, \mathbf{y}, \widetilde{\mathbf{X}}) = \mathcal{N}(\mathbf{K}_M^\theta (\mathbf{Q}_M^\theta)^{-1}\mathbf{K}_{MN}^\theta (\mathbf{\Lambda} + \sigma^2\mathbb{1})^{-1}\mathbf{y}, \ \mathbf{K}_M^\theta (\mathbf{Q}_M^\theta)^{-1}\mathbf{K}_M^\theta), \tag{2.12}$$

where $\mathbf{Q}_M^\theta = \mathbf{K}_M^\theta + \mathbf{K}_{MN}^\theta (\mathbf{\Lambda} + \sigma^2\mathbb{1})^{-1}\mathbf{K}_{NM}^\theta$. We assume for now, that the pseudo inputs $\widetilde{\mathbf{X}}$ and (locally optimized) hyperparameters $(\theta_\star, \sigma_\star)$ are given. Therefore, we can make predictions for a new data point $\mathbf{x}^* \in \mathcal{X}$, with noisy target $y^* = f(\mathbf{x}^*) + \varepsilon$, with $\varepsilon \sim \mathcal{N}(0, \sigma_\star^2)$, by using the predictive distribution

$$\begin{aligned}
&\mathcal{P}_{\theta_\star,\sigma_\star}(y^*|\mathbf{x}^*, \mathbf{X}, \mathbf{y}, \widetilde{\mathbf{X}}) = \mathcal{N}(\mu_*, \varsigma_*^2), \\
&\mu_* := K_{\theta_\star}(\widetilde{\mathbf{X}}, \mathbf{x}^*)^\top (\mathbf{Q}_M^{\theta_\star})^{-1}\mathbf{K}_{MN}^{\theta_\star}(\mathbf{\Lambda} + \sigma_\star^2\mathbb{1})^{-1}\mathbf{y}, \\
&\varsigma_*^2 = K_{\theta_\star}(\mathbf{x}^*, \mathbf{x}^*) + K_{\theta_\star}(\widetilde{\mathbf{X}}, \mathbf{x}^*)^\top ((\mathbf{K}_M^{\theta_\star})^{-1} - (\mathbf{Q}_M^{\theta_\star})^{-1})K_{\theta_\star}(\widetilde{\mathbf{X}}, \mathbf{x}^*) + \sigma_\star^2.
\end{aligned} \tag{2.13}$$

Since the matrix $\mathbf{\Lambda} + \sigma^2\mathbb{1}$ is diagonal its inversion costs are negligible, whereas inversion of the lower dimensional matrices $\mathbf{K}_M^{\theta_*}$ and $\mathbf{Q}_M^{\theta_\star}$ of order $O(M^3)$ such that the total costs are now dominated by the matrix multiplication costs of order $O(MN^2)$ that are required to obtain $\mathbf{Q}_M^\star$. In total the training time can be reduced from $O(N^3)$ to $O(M^2N)$, which is a highly significant advantage for small M compared to standard GP.

As for the standard GP, the marginal likelihood of the noisy observations $\mathbf{y}$ can be used to determine (locally) optimal hyperparameters $(\theta_\star, \sigma_\star)$. Given Equation (2.10) and Equation (2.11), the marginal likelihood becomes

$$p_{\theta,\sigma}(\mathbf{y}|\mathbf{X}, \widetilde{\mathbf{X}}) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_{NM}^\theta (\mathbf{K}_M^\theta)^{-1}\mathbf{K}_{MN}^\theta + \mathbf{\Lambda} + \sigma^2\mathbb{1}). \tag{2.14}$$

In contrast to Equation (2.7), the true kernel is replaced by an approximated pseudo input dependent kernel in Equation (2.14) sharing the same diagonal entries. The inducing points $\widetilde{\mathbf{X}}$ must therefore be considered as additional kernel hyperparameters that need to be fitted additionally to the kernel parameters in Equation (2.10). As a result, we need to jointly optimize the much larger

set of hyperparameters $\{\widetilde{\mathbf{X}}, \theta, \sigma\}$. SGP therefore trades off reduced computational complexity for the determination of the posterior with increased hyperparameter tuning costs, with the latter being a common source of overfitting when simple gradient descent methods are used.

To overcome this issue, for example, Titsias (2009) propose a variational approach to maximize an appropriate lower bound of the true marginal likelihood, thereby treating $\widetilde{\mathbf{X}}$ as variational hyperparameters instead of additional kernel hyperparameters.

Acquisition Function

The second component for Bayesian optimization is the acquisition function. The acquisition function trades off exploration of regions where our probabilistic model is uncertain and exploitation regions, where our probabilistic model expects a high objective function value. The most popular functions are:

Expected Improvement The expected improvement acquisition function (EI) (Mockus et al., 1978; Jones et al., 1998) is the most used one and is defined as:

$$a^{\text{EI}}_{\mathcal{P}(f|D_N)}(\mathbf{x}) = \mathbb{E}_{\mathcal{P}(f|D_N)}[\max(f(\mathbf{x}) - y^+, 0)], \tag{2.15}$$

with $y^+ = \max(y_0, \ldots, y_N)$ being the best observed function value at time N. In case of a Gaussian posterior distribution $\mathcal{P}(f|D_N)$, for example if the probabilistic function is a Gaussian process, the expected improvement can be calculated in a closed form (Brochu et al., 2010):

$$\begin{aligned} a^{\text{EI}}_{\mathcal{P}(f|D_N)}(\mathbf{x}) &= (\mu(\mathbf{x}) - y^+)\Phi(Z) + \sigma(\mathbf{x})\phi(Z), \\ Z &= \frac{\mu(\mathbf{x}) - y^+}{\sigma(\mathbf{x})} \end{aligned} \tag{2.16}$$

with $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ being the mean and standard deviation of the probabilistic model at $\mathbf{x}$, and $\phi(\cdot)$ and $\Phi(\cdot)$ being the PDF and CDF of the standard normal distribution.

Upper Confidence Bound Upper confidence bound (UCB)(Cox and John, 1992; Brochu et al., 2010) computes:

$$a^{\text{UCB}}_{\mathcal{P}(f|D_N)}(\mathbf{x}) = \mu(\mathbf{x}) + \theta\sigma(\mathbf{x}), \tag{2.17}$$

with $\theta \geq 0$ being an exploration-exploitation hyperparameter.

Probabilistic Improvement Probabilistic improvement (PI) (Jones et al., 1998) calculates the probability of exceeding the current best function value y^+:

$$a^{\text{PI}}_{\mathcal{P}(f|D_N)}(\mathbf{x}) = \Phi(y^+). \tag{2.18}$$

2.3.2 *Variational Autoencoder*

In this section, we will provide an in-depth overview of *variational autoencoder* (VAE) as presented by Kingma and Welling (2014).

Given i.i.d. data $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^N \in \mathcal{X}$, we are interested in learning a continuous latent space $\mathcal{Z} \in \mathbb{R}^J, J \ll d$, involving a dimensionality reduction. This process consists of a generative

model $p_\theta(\mathbf{x}|\mathbf{z})$ implemented as a neural network, given some prior distribution $p_\theta(\mathbf{z})$ and a recognition model $p_\theta(\mathbf{z}|\mathbf{x})$, which we also implement as a neural network. Using the *Bayes' rule*, the recognition model (posterior) can be written as:

$$p_\theta(\mathbf{z}|\mathbf{x}) = \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{x})}, \tag{2.19}$$

with the margin likelihood $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})dz$ being intractable in the case of neural networks. This leads in total to an intractable true posterior.

In order to solve the intractability, Kingma and Welling (2014) make use of variational inference (see Blei et al. (2017) for a review), which can be posed as the problem of finding a model $q_\phi(\mathbf{z}|\mathbf{x})$ which approximates the true intractable posterior $p_\theta(\mathbf{z}|\mathbf{x})$:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \arg\min_q D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})), \tag{2.20}$$

with D_{KL} being the Kullback–Leibler divergence (Sullivan, 2015).

Definition 3 (Kullback–Leibler Divergence). *The Kullback–Leibler divergence D_{KL} between two densities $q\colon \mathbb{R}^n \to [0,\infty)$ and $p\colon \mathbb{R}^n \to [0,\infty)$ is defined as:*

$$\begin{aligned} D_{KL}(q\|p) &= \int q(z)\log\frac{p(z)}{q(z)}dz \\ &= \mathbb{E}_{q(z)}\left[\log\frac{p(z)}{q(z)}\right], \end{aligned} \tag{2.21}$$

with $\mathbb{E}_{q(z)}[f(z)] \coloneqq \int f(z)q(z)dz$.

Rewriting Equation (2.20) using Definition 3 leads to:

$$\begin{aligned} D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x})}\right] \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x})] - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})] \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x})] - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z},\mathbf{x})] + \log p_\theta(\mathbf{x}). \end{aligned} \tag{2.22}$$

Therefore, we can rewrite the Kullback–Leibler divergence in terms of the marginal log-likelihood $\log p_\theta(\mathbf{x})$, and thus rewrite the latter as following:

$$\begin{aligned} \log p_\theta(\mathbf{x}) &= D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x})] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z},\mathbf{x})] \\ &\geq -\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x})] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z},\mathbf{x})] \\ &= -\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x})] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z})] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z})), \end{aligned} \tag{2.23}$$

with $D_{KL} \geq 0$ by definition. The right-hand side of the last equation of Equation (2.23) is known as the *variational lower bound (ELBO)* on the marginal likelihood. Therefore, the intractable marginal likelihood can now be approximated by its variational lower bound, which we write as:

$$\mathcal{L}(\theta,\phi;\mathbf{x}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z})). \tag{2.24}$$

In total, for a variational autoencoder with a probabilistic encoder model $q_\phi(\mathbf{z}|\mathbf{x})$ and a probabilistic decoder $p_\theta(\mathbf{x}|\mathbf{z})$, the goal is to differentiate and optimize the variational lower bound w.r.t both ϕ and θ. However, differentiating the lower bound w.r.t ϕ in $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ is problematic since that would involve differentiation through a sample $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$. Therefore, an efficient gradient estimator as Monte Carlo is needed:

$$\nabla_\phi \mathbb{E}_{q_\phi(\mathbf{z})}[f(\mathbf{z})] = \mathbb{E}_{q_\phi(\mathbf{z})}[f(\mathbf{z})\nabla_{q_\phi(\mathbf{z})} \log q_\phi(\mathbf{z})] \simeq \frac{1}{L}\sum_{l=1}^{L} f(\mathbf{z})\nabla_{q_\phi(\mathbf{z}_l)} \log q_\phi(\mathbf{z}_l), \tag{2.25}$$

with $\mathbf{z}_l$ i.i.d samples from $q_\phi(\mathbf{z}|\mathbf{x}_l)$. As mentioned in Kingma and Welling (2014), the variance of this gradient estimator can become very large, which makes it an impractical estimator. As a result, Kingma and Welling (2014) introduce a so-called reparameterization trick: Instead of sampling $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$, the random variable $\mathbf{z}$ will be expressed as a deterministic variable, using a vector-valued function $g_\phi(\cdot)$ parameterized by ϕ and an auxiliary variable ε with independent marginal $p(\varepsilon)$:

$$\mathbf{z} = g_\phi(\varepsilon, \mathbf{x}). \tag{2.26}$$

This way, the Monte Carlo estimate of the expectation, $\frac{1}{L}\sum_{l=1}^{L} f(g_\phi(\varepsilon_l, \mathbf{x}))$, with i.i.d. $\varepsilon_l \sim p(\varepsilon)$, is differentiable w.r.t ϕ. Note, $L = 1$ mostly in practice.

After introducing the essential steps in a variational autoencoder, we will describe the most commonly used example of a Gaussian variational autoencoder. The prior is then defined as a multivariate Gaussian $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbb{1})$ and the decoder model $p_\theta(\mathbf{x}|\mathbf{z})$ is also assumed to be a multivariate Gaussian. Furthermore, the probabilistic encoder model $q_\phi(\mathbf{z}|\mathbf{x})$ is assumed to be approximate Gaussian with a diagonal covariance matrix, i.e., has density $\mathcal{N}(\mathbf{z}|\mu(\mathbf{x}), \sigma^2(\mathbf{x})\mathbb{1})$, where the mean μ and the variance σ^2 are outputs of the neural network. As previously discussed, we now use the reparameterization trick for a sample $\mathbf{z_{n,l}}$.

$$\mathbf{z_{n,l}} = g_\phi(\varepsilon_l, \mathbf{x}_n) = \mu(\mathbf{x}_n) + \sigma(\mathbf{x}_n)\varepsilon_l,$$

with $\varepsilon_l \sim \mathcal{N}(\mathbf{0}, \mathbb{1})$.

Since both models $q_\phi(\mathbf{z}|\mathbf{x}_n)$ and $p_\theta(\mathbf{z})$ are Gaussian, the Kullback–Leibler divergence can be computed in a closed form. Overall, given a sample $\mathbf{x}_n$, the resulting objective from Equation (2.24) takes the form

$$\mathcal{L}(\theta, \phi; \mathbf{x}_n) = \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(\mathbf{x}_n|\mathbf{z_{n,l}}) + \frac{1}{2}\sum_{j=1}^{J}\left(1 + \log \sigma_j(\mathbf{x}_n)^2 + \mu_j(\mathbf{x}_n)^2 - \sigma_j(\mathbf{x}_n)^2\right), \tag{2.27}$$

with J being the dimensionality of $\mathbf{z}$, and $\mu(\mathbf{x}_n) = (\mu_j(\mathbf{x}_n))_{j=1,\dots,J}$ and $\sigma(\mathbf{x}_n) = (\sigma_j(\mathbf{x}_n))_{j=1,\dots,J}$ denote the variational mean and standard deviation evaluated at $\mathbf{x}_n$, respectively. The parameters θ and ϕ are optimized by maximizing Equation (2.27). Summarize again, the first term of Equation (2.27) is the reconstruction loss and enforces high similarity between input data and generated data, while the second term is the Kullback–Leibler divergence, which regularizes the latent space. From the trained VAE, new data can be generated by simply decoding latent space variables $\mathbf{z}$ sampled from the prior distribution $p(\mathbf{z})$.

Algorithm 2: Bayesian Optimization in the Latent Space

Input: (i) data $D_{N-1} = \{(\mathbf{x}_n, f(\mathbf{x}_n))\}_{n=1}^{N-1}$
Input: (ii) objective function f
Input: (iii) latent objective model h
Input: (iv) acquisition function a
Input: (v) data query budget T
Input: (vi) generative and inverse model p, q trained on $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$
for $N \leq T$ **do**
  compute latent variables $\mathcal{Z}_{\mathbf{X}} = \{q(\mathbf{x}) = \mathbf{z}\}_{\mathbf{x} \in D_{N-1}}$
  fit latent objective model h on data $\mathcal{Z}_{\mathbf{X}}$ and D_{N-1}
  use acquisition function to sample $\widetilde{\mathbf{z}} = \arg\max_{\mathbf{z} \in \mathbb{R}^J} a_h(\mathbf{z})$
  obtain $\mathbf{x}_N = p(\widetilde{\mathbf{z}})$ and evaluate $f(\mathbf{x}_N)$
  augment data $D_N \leftarrow D_{N-1} \cup (\mathbf{x}_N, y_N)$
  $N \leftarrow N + 1$
end

Optimization in the Latent Space

We now provide more information about how a latent space, especially one generated by a VAE, can be further used for optimization approaches such as Bayesian optimization.

Optimization in the latent space has emerged in several tasks in recent years, as molecular designs (Gómez-Bombarelli et al., 2018; Kusner et al., 2017; Jin et al., 2018) and NAS (Luo et al., 2018; Zhang et al., 2019; Yan et al., 2020). This optimization is a two-step procedure (Tripp et al., 2020): first, we learn a mapping from a continuous latent space $\mathcal{Z} \in \mathbb{R}^J$ to the original data space $\mathcal{X}$ using a generative model, e.g., the decoder from the previously introduced VAE, $p : \mathcal{Z} \to \mathcal{X}$ Second, we learn a latent objective model $h : \mathcal{Z} \to \mathbb{R}$, which approximates the black-box-function $f : \mathcal{X} \to \mathbb{R}$ as a surrogate model at the output of p, s.t. $f(p(\mathbf{z})) \approx h(\mathbf{z}), \forall \mathbf{z} \in \mathcal{Z}$. This latent objective model can be learned by using an inverse of the generative model (the encoder model in a VAE setting), i.e., $q : \mathcal{X} \to \mathcal{Z}$. In the case of applying Bayesian optimization in this setting, the latter latent objective function h is commonly a Gaussian process, as introduced in Section 2.3.1. An overview is provided in Algorithm 2.

Part I

Performance Estimation Strategy

Graph Neural Network-based Prediction Model

3

The overwhelming success of convolutional neural architectures is due to the increasing availability of training data and compute resources, which have ultimately allowed the design of new architectures that surpass human performance. Neural Architecture Search, as introduced in Section 2.1, is the next step in automating the process of topology design of architectures. Of particular note is the publication of the benchmark NAS-Bench-101 (Ying et al., 2019), which is the foundation for this chapter.

In this chapter, we first tackle the task of learning to predict the accuracy of convolutional neural architectures in a supervised way, i.e., we learn a surrogate model that enables to predict the performance of neural architectures on the CIFAR-10 image classification task (Krizhevsky, 2009). Furthermore, we evaluate our proposed model on two different zero-shot prediction scenarios and show its ability to accurately predict performances in previously unseen regions of the search space. Therefore, this chapter contributes to the NAS research part *performance estimation strategy*.

Most current neural architectures for computer vision can be represented as directed acyclic graphs (DAGs). Thus, using Graph Neural Networks (GNNs) as introduced in Section 2.2 is an obvious choice. The capability of GNNs to accurately model dependencies between nodes makes them the foundation of this chapter. We utilize them to move from the discrete graph space to the continuous space. More precisely, we show the ability of GNNs to encode neural architectures from NAS-Bench-101 (Ying et al., 2019) such as to allow for a regression of their expected performance. The ability of GNNs to comprehend local node features and graph substructures makes them a very useful tool to embed nodes as well as complete graphs like the NAS-Bench-101 architectures into continuous spaces. Furthermore, the benefit of GNNs over Recurrent Neural Networks (RNNs) has been shown in the context of graph generating models. The model *Deep Generative Models of Graphs* (DGMG) (Li et al., 2018b) utilizes GNNs and shows dominance over RNN methods. DGMG can capture the structure of graph data and its attributes so that probabilistic dependencies within graph nodes and edges can be expressed, yielding in learning a distribution over any graph. This makes DGMG a strong tool for mapping neural architectures into a feature representation that captures the complex relation within the neural architecture. Inspired by the GNN in Li et al. (2018b), we utilize it as our surrogate model for the performance prediction task.

In summary, in this chapter, we make the following contributions:

- We present a surrogate model– a graph encoder built on GNNs– for neural architecture performance prediction trained and evaluated on NAS-Bench-101 architectures with the target to predict the validation and test accuracy.
- We show that this performance predictor accurately predicts architecture performances in previously structurally different and unseen regions of the search space, i.e., our model enables zero-shot prediction.

The remaining chapter is structured as follows: We present our proposed encoder model in Section 3.1. In Section 3.2, we present our experiments and results. Finally, we give a conclusion and outline future directions in Section 3.3.

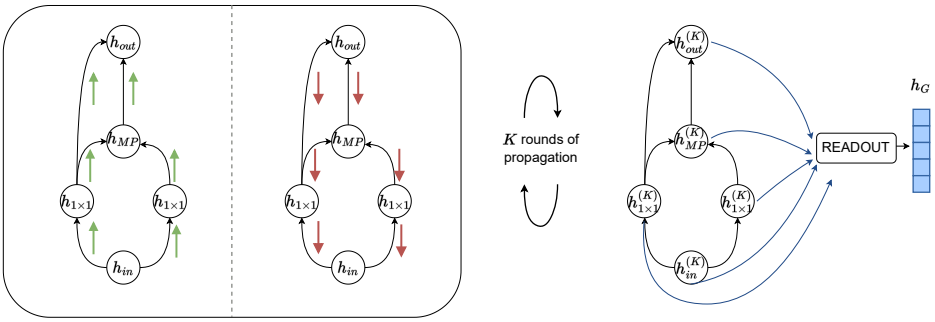


Figure 3.1: Illustration of the graph encoding process. **(left)** The node-level propagation using T rounds of bidirectional message passing. **(right)** The graph-level aggregation into a single graph embedding $\mathbf{h}_G$.

This chapter is a slightly modified version of the paper J. Lukasik et al. (2020a). "Neural Architecture Performance Prediction Using Graph Neural Networks". In: *Proc. of the German Conference on Pattern Recognition (GCPR)*. The code for this chapter is available on GitHub[1].

3.1 The Graph Encoder

In this section we present our GNN-based model to encode the discrete graph space of NAS-Bench-101 (Ying et al., 2019) into a continuous vector space. We find that the settings similar to Li et al. (2018b) work best for our needs.

3.1.1 *Node-Level Propagation*

As introduced in Section 2.2.1, we define a neural network as a DAG $G = (V, E)$. Here, we furthermore denote $\mathcal{V}^{(out)}(v) = \{u \in V \mid (v,u) \in E\}$ as an additional set of adjacent nodes to node $v \in V$ with outgoing edges.

For each node $v \in V$, we associate an initial node embedding $\mathbf{h}_v^{(0)} \in \mathbb{R}^{d_n}$. Instead of using one-hot encoded node labels in combination with an additional neural network to learn these initial node embeddings, we employ a learnable embedding table L_e on the node types (i.e., indices of the one-hot encoded node labels), which stores embeddings for our initial node embeddings $\mathbf{h}_v^{(0)}$.

The aggregation module $\mathcal{A}$ as presented in Section 2.2.1 Equation (2.2) is in this section formulated as:

$$\begin{aligned}\mathbf{a}_v^{(k)} &= \mathcal{A}^{(k)}\left(\left\{\mathbf{h}_u^{(k-1)} : u \in \mathcal{V}(v)\right\}\right) \\ &= \sum_{u\in\mathcal{V}(v)} f\left(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}\right) \\ &= \sum_{u\in\mathcal{V}(v)} \mathrm{fc}\Big(\mathrm{CONCAT}\Big(\left[\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}\right]\Big)\Big),\end{aligned} \tag{3.1}$$

[1] https://github.com/jovitalukasik/GNN_predictor

where fc is a single fully connected layer. The message passing is illustrated by the green arrows in Figure 3.1 (left). In addition, we add a reverse message module:

$$\begin{aligned} \mathbf{a}^{(out)}{}_v^{(k)} &= \sum_{u \in \mathcal{V}^{(out)}(v)} f^{(out)}\left(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}\right) \\ &= \sum_{u \in \mathcal{V}^{(out)}(v)} \mathrm{fc}^{(out)}\left(\mathrm{CONCAT}\left(\left[\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}\right]\right)\right), \end{aligned} \tag{3.2}$$

with $\mathrm{fc}^{(out)}$ being also a single fully connected layer.

The reverse message passing is outlined in Figure 3.1 (left) by the red arrows and leads to so-called bidirectional message passing. This step allows us to not only aggregate incoming node information, but also aggregate outgoing information, thus providing more details on the information flow in the graph. This ultimately leads to better information aggregation.

Including both directions of aggregations leads to the following update module $\mathcal{U}^{(k)}$ based on Equation (2.3):

$$\mathbf{h}_v^{(k)} = \mathcal{U}^{(k)}\left(\mathbf{h}_v^{(k-1)}, \mathbf{a}_v^{(k)} + \mathbf{a}^{(out)}{}_v^{(k)}\right), \tag{3.3}$$

where the second term $\mathbf{a}_v^{(k)} + \mathbf{a}^{(out)}{}_v^{(k)}$ aggregates the messages in both directions, forward propagation as in the directed graph as well as the reverse propagation, at each node v. We set $\mathcal{U}$ as a single gated recurrent unit (GRU) (Cho et al., 2014) where $\mathbf{h}_v^{(k-1)}$ is treated as the hidden state.

3.1.2 *Graph-Level Aggregation*

After the final round of message passing, the propagated node embeddings $\mathbf{h}_v^{(K)}$ are aggregated into a single graph embedding $\mathbf{h}_G \in \mathbb{R}^{d_g}$ as presented Equation (2.4). We set READOUT as:

$$\mathbf{h}_G = \sum_{v \in V} \mathrm{fc}\left(\mathbf{h}_v^{(K)}\right) \odot \sigma\left(\mathrm{fc}\left(\mathbf{h}_v^{(K)}\right)\right) \tag{3.4}$$

with fc being a new single fully connected layer, $\sigma(\mathrm{fc}(\cdot))$ being a separate gating layer, which is a logistic sigmoid function combined with a fully connected layer, that adjusts each node's fraction in the graph embedding, and $\odot$ denoting the Hadamard product. This graph aggregation layer in Equation (3.4) is further illustrated in Figure 3.1 (right).

3.2 Experiments

We conduct experiments in three complementary domains. First, we evaluate the performance prediction ability of the proposed GNN in the traditional supervised setting. Then, we conduct zero-shot prediction experiments in order to show the performance of the proposed model for unseen graph structures during training. Both experiments are carried out on the validation accuracies reported in NAS-Bench-101. Last, we compare our results to the recent publication by Tang et al. (2020) in terms of test accuracy prediction.

Implementation Details.

We split the architectures in NAS-Bench-101 in 70%/ 20%/ 10% edit-sampled into training-, test- and validation set. We provide further visualization of the NAS-Bench-101 search space in Appendix A.1, and further detailed information about the used hyperparameters in Appendix B.1.

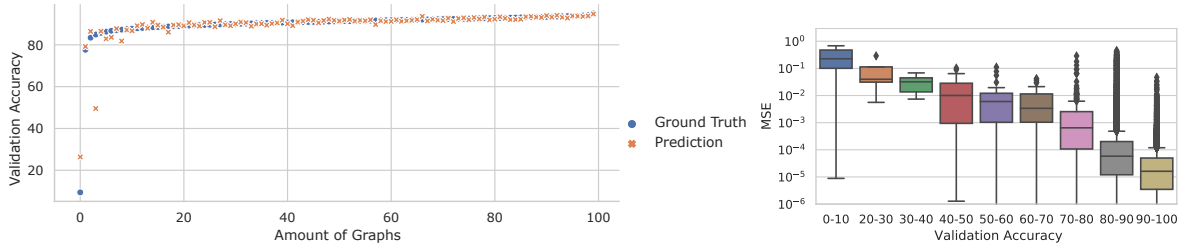


Figure 3.2: (**left**) The predicted and ground truth validation accuracy (in %) of 100 randomly sampled graphs from NAS-Bench-101 show a low prediction error for graphs with high accuracy. For low-accuracy architectures, our model mostly predicts low values. (**right**) The mean and variance of the squared error of the test set performance prediction sorted by the ground truth validation accuracy (in %) in logarithmic scale. Predictions are very reliable for architectures in the high accuracy domain, while errors are higher for very low-performing architectures.

Surrogate Model	**Supervised Performance Prediction** (RMSE in 10^{-2}) ↓
Random Forest Width-Depth Feature Encoding	6.10 ± 0.40%
Random Forest One-Hot Encoding	6.32 ± 0.01%
MLP One-Hot Encoding	6.32 ± 0.02%
RNN One-Hot Encoding	6.30 ± 0.01%
GNN Encoder (ours)	**4.86** ± 0.10%

Table 3.1: Predictive performance of the GNN encoder in terms of RMSE on supervised validation performance prediction on NAS-Bench-101.

3.2.1 *Performance Prediction*

Supervised Performance Prediction.

Here, we evaluate the latent space generated by the encoder with respect to its prediction error regarding a metric of interest of the NAS-Bench-101 graphs, i.e., the validation accuracy on CIFAR-10 (Krizhevsky, 2009). For this purpose, we utilize a simple predictor, i.e., a four-layer Multilayer Perceptron (MLP) with ReLU non-linearities.

We jointly train the encoder and the predictor supervisedly end-to-end. We test for prediction as well as for zero-shot prediction errors. There are a few outliers in the NAS-Bench-101 graphs that end up with a low validation accuracy on the CIFAR-10 classification task. Figure 3.2 (left) visualizes these outliers and shows that our model is able to find them even if it cannot perfectly predict their accuracies. One can see that the model very accurately predicts the validation accuracy of well-performing graphs. To further explore the loss, Figure 3.2 (right) illustrates the mean and variance of the squared error of the test set partitioned in 9 bins with respect to the ground truth accuracy of the complete test set. The greater part of the loss arises from graphs with low accuracy. More importantly, our model is very accurate in predicting graphs of interest, namely graphs with high accuracy.

The rather bad prediction of graphs with low and intermediate accuracy can be explained through their low share in the search space, as discussed in Section 2.1.6 and Figure 2.5. Taking a look at the distribution of the individual accuracies in the overall NAS-Bench-101 search space,

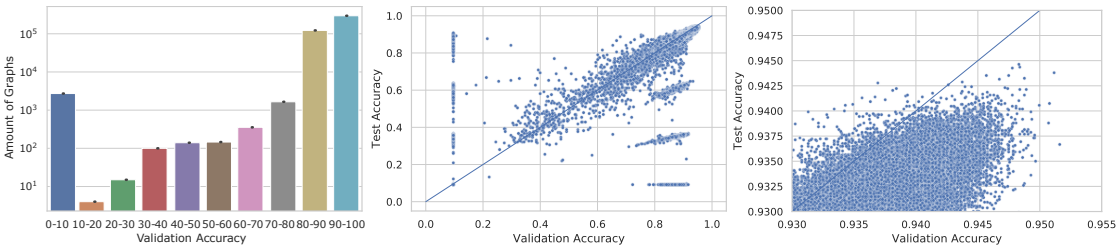


Figure 3.3: Distinct properties of NAS-Bench-101. **(left)** The allocation of the architectures sorted by the ground truth accuracy in % in logarithmic scale ~98.8% in the two last bins. NAS-Bench-101 validation and test accuracy behavior on the CIFAR-10 image classification task. **(middle)** Validation accuracy in % compared to the test accuracy in % of the neural architectures in NAS-Bench-101. **(right)** A more precise look into the areas of interest for neural architectures displays that the best neural architecture by means of the test accuracy is unequal to the best accuracy by means of the validation accuracy.

Surrogate Model	**Zero-Shot Performance Prediction** (RMSE in 10^{-2}) ↓	
	$2, 3, 4, 5, 7 - 6$	$2, 3, 4, 5, 6 - 7$
Random Forest Width-Depth Feature Encoding	6.00 ± 0.20%	7.30 ± 0.5%
Random Forest One-Hot Encoding	7.00 ± 0.04%	6.30 ± 0.1%
MLP One-Hot Encoding	6.47 ± 2.40%	9.40 ± 12.7%
RNN One-Hot Encoding	6.20 ± 4.70%	6.90 ± 3.3%
GNN Encoder (ours)	**5.23** ± 3.90%	**5.73** ± 1.7%

Table 3.2: Predictive performance of the GNN encoder in terms of RMSE on the two different zero-shot validation performance prediction tasks on NAS-Bench-101.

as shown in Figure 3.3 (left), illustrates the low share of low and intermediate accuracies and explains, therefore, the rather bad prediction behavior of our surrogate model. Figure 3.3 (middle) and Figure 3.3 (right) plot the validation accuracy compared to the test accuracy in NAS-Bench-101. This figure illustrates that predicting the best architecture on the validation set does not necessarily imply a proper prediction on the test set.

We compare the results of the encoder to several baselines. Our baselines are a random forest approach and also an MLP regression model with four layers, using one-hot node feature encodings and graph depth/width feature encodings. In order to compare to an RNN baseline, we adapted the RNN-surrogate model from Liu et al. (2018a), which only handles cells of equal length. For the application to NAS-Bench-101 with cell types of different lengths, we modify the node attributes as one-hot vector encodings with 0-padding to fill up to seven nodes and five operations.

Table 3.1 summarizes the performance prediction results on the supervised performance prediction task. All experiments are repeated 3 times, and we report the mean and the relative standard deviation. This experiments show that our surrogate model is able to predict the neural architecture performances stably and outperforms all baselines in terms of RMSE, which denotes the square root of the empirical squared loss between the predicted and ground truth data (MSE), by a significant margin.

Zero-shot Performance Prediction.

Next, we consider the task of predicting the validation accuracy of structurally unknown graph types, i.e., zero-shot prediction. The zero-shot prediction task is furthermore divided into two subtasks. First, the encoder is trained on all graphs of length 2, 3, 4, 5, 7 and tested on graphs of length 6. In this scenario, the unseen architectures could be understood as interpolations of seen architectures. Second, we learn the encoder on graphs of length 2, 3, 4, 5, 6 and test it on graphs of length 7. This case is expected to be more challenging not only because the graphs of length 7 are the clear majority and have the highest diversity but also because the prediction of their performance is an extrapolation out of the seen training distribution.

Table 3.2 summarizes the performance prediction results on the zero-shot performance prediction task. All experiments are repeated 3 times, and we report the mean and the relative standard deviation. As expected, the resulting RMSE is slightly higher for the extrapolation to graphs of length 7 than for the zero-shot prediction for graphs of length 6. Yet, the overall prediction improves over all baselines by a significant margin. The higher standard deviation compared to the random forest baselines indicates that the performance of the GNN depends more strongly on the weight initialization than in the fully supervised case. Yet, please note that this dependence on the initialization is still significantly lower than for the MLP and RNN baselines. The experiments show that our surrogate model is able to accurately predict data that it has never seen, i.e., that it can predict the accuracies even for architectures not represented by the training distribution.

3.2.2 *Training Behavior*

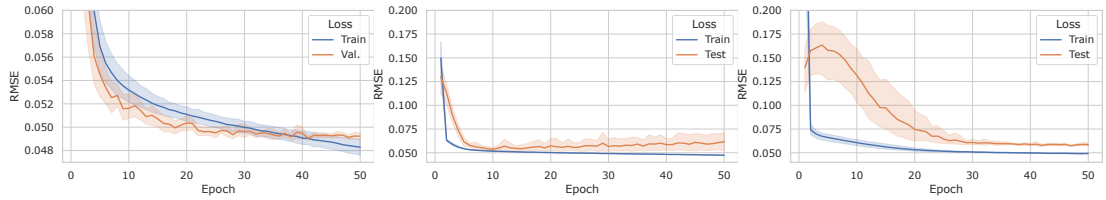
In the following, we analyze the training behavior of our model in the different scenarios described above.

Supervised Performance Prediction.

For visualization aspects of the training behavior of our encoder, we plot the development of the training loss against the validation loss for the supervised performance prediction from Section 3.2.1. Figure 3.4 (left) displays this development of training loss against validation loss measured by means of the RMSE. The smallest achieved RMSE is 0.0487 for training on 70% of all architectures, i.e., 296 558 samples.

Zero-shot Performance Prediction.

The progress of the training loss and test error for the *zero-shot prediction* case of our encoder can be seen in Figure 3.4 (middle, right). The training set containing all graphs of length 2, 3, 4, 5, 7 / 2, 3, 4, 5, 6 has a total amount of 361 614/64 542 samples. Thus, the encoder is tested only on graphs of length 6/7, corresponding to a total of 62 010/359 082 neural architectures. The experiments show that our model is able to accurately predict data that it has never seen before. The behavior of the test error during the second zero-shot prediction task, see Figure 3.4 (right), displays interesting information. During the first epochs, the error rises before it starts decreasing and approaching the training loss asymptotically. One interpretation could be that the model first



(a) Supervised. (b) Zero-shot 6. (c) Zero-shot 7.

Figure 3.4: Progress of loss and validation error over 50 epochs regarding performance prediction. **(left)** Supervised performance prediction task with best validation RMSE of 0.0487. **(middle)** Zero-shot performance prediction with training set containing of all graphs of length 2, 3, 4, 5, 7 with a test set of the graphs of length 6. **(right)** Zero-shot performance prediction with training set containing all graphs of length 2, 3, 4, 5, 6 with a test set of the graphs of length 7.

Model	**Performance Prediction** (MSE in 10^{-3}) ↓		
	1 000	10 000	100 000
Semi-Supervised Assessor (Tang et al., 2020)	**3.1** $\pm$ 3×10^{-4}	2.6 $\pm$ 2×10^{-4}	1.6 $\pm$ 2×10^{-4}
GNN Encoder (ours)	4.4 $\pm$ 3×10^{-4}	**2.2** $\pm$ 1×10^{-4}	**1.5** $\pm$ 1×10^{-5}

Table 3.3: Comparison of the predictive performance of surrogate models in terms of MSE on the test accuracies in NAS-Bench-101.

learns simple graph properties like the number of nodes before it learns more complex graph substructures that generalize to the unseen data.

3.2.3 *Comparison to State of the Art*

This section compares our GNN-surrogate model with the state-of-the-art predictor from Tang et al. (2020), which evaluates the predictor on the test accuracy of NAS-Bench-101. Since predicting the validation accuracy does not imply the same proper prediction behavior on the test set, we evaluate our surrogate model in the same setting. In Tang et al. (2020), an auto-encoder model is first trained on the entire NAS-Bench-101 data and then fine-tuned with a graph similarity metric and test accuracy labels. Because the training relies on unsupervised pretraining, they refer to the approach as semi-supervised. To enable a direct comparison, we randomly sample 1 000/10 000/100 000 graphs from the training data set and evaluate the performance prediction ability of the GNN surrogate model on all remaining 431 624/413 624/323 624 graphs in NAS-Bench-101. Please note that, at training time, the semi-supervised approach from Tang et al. (2020) actually has access to more data than our fully supervised approach because of the unsupervised pretraining.

Table 3.3 shows the experimental comparison, where we report an average over three runs for our approach while the numbers of Tang et al. (2020) are taken from their paper. The proposed GNN surrogate model surpasses the proposed semi-supervised assessor (Tang et al., 2020) when 10 000 and 100 000 training architectures are available. With only 1 000 randomly drawn training samples, the results of our approach decrease. Yet, since we do not have access to the exact training samples used in Tang et al. (2020), the results might become less comparable the lower the number of samples drawn.

3.3 CONCLUSION

In this chapter, we propose a GNN surrogate model for predicting the performance of neural architectures. Through multiple experiments on NAS-Bench-101, we examine various capabilities of the encoder. The GNN encoder is a powerful tool for supervised performance prediction, especially in the zero-shot setup. Further research will mainly review the possibilities of neural architecture search in accordance with further performance prediction. We tackle this possibility in the following chapter Chapter 4.

Part II

Generative Architecture Search

Variational Autoencoder-based Graph Embeddings

4

In the previous chapter, we introduced a performance estimation surrogate model that enables to accurately predict the performance of neural architectures in two distinct settings, classical prediction and zero-shot prediction. Surrogate models are a powerful tool for the search strategy in Neural Architecture Search (White et al., 2021c). Initial NAS methods using reinforcement learning (Zoph and Le, 2017; Zoph et al., 2018), evolutionary algorithm (Elsken et al., 2019; Real et al., 2019) or Bayesian optimization (BO) (Kandasamy et al., 2018; White et al., 2021a; Ru et al., 2021) typically required thousands of function evaluations to find a good solution, which is infeasible without company-scale compute infrastructure. Therefore, recent research in NAS focuses on efficient methods via continuous relaxations of the discrete search space and weight-sharing approaches (Bender et al., 2018; Pham et al., 2018; Liu et al., 2019; Cai et al., 2019), with the possible drawback of sub-optimal results (Zela et al., 2020a), as discussed in Section 2.1.4. As presented in Chapter 3, GNNs (Gori et al., 2005; Kipf and Welling, 2017; Wu et al., 2021c) are a very useful and suitable tool to embed local node features and graph substructures. Therefore, we argue in this chapter in favor of NAS on learned graph embeddings using GNNs on the encoder and decoder level, and contribute to the NAS search method research. Zhang et al. (2019) showed good performance with such a model, called D-VAE, on the ENAS search space (Pham et al., 2018) for neural architecture performance prediction and BO - proving its ability to learn smooth continuous graph representations. D-VAE aggregates information in the GNN alternatingly in the forward and backward directions to encode the neural network information flow. However, the D-VAE model imposes strong constraints on the graph structure, which limit its applicability to search spaces beyond ENAS. In addition, it has very long training times.

In this chapter, we propose a two-sided variational GNN encoder-decoder to learn smooth embeddings in various NAS search spaces, which we call *Smooth Variational Graph embedding* (SVGe). In contrast to D-VAE, SVGe aggregates node representations in the forward and backward direction separately and consequently decodes their joint representation into forward and backward direction separately (see Figure 4.1). This yields a very high reconstruction ability without imposing any constraints on the search space and allows for more efficient training.

Inheriting from variational autoencoders (Kingma and Welling, 2014) (Section 2.3.2), it focuses on learning an accurate architecture mapping into a smooth latent space in which structurally similar graphs are close to one another and thus facilitates efficient black-box optimization to find high-performing architectures. The proposed model is not only three times faster than D-VAE but also shows improved BO results in the ENAS search space. In contrast to D-VAE, it can be directly applied to other search spaces such as NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201 (Dong and Yang, 2020). It also allows competitive performance to highly optimized approaches using BO (Yan et al., 2020).

Moreover, it allows to learn architecture performance prediction in a supervised way. Due to its smoothness, it can further directly extrapolate the performance prediction outside the space of seen architectures. Thus, it proposes high-performing, deeper networks without using BO at very low costs.

In summary, we make the following contributions:

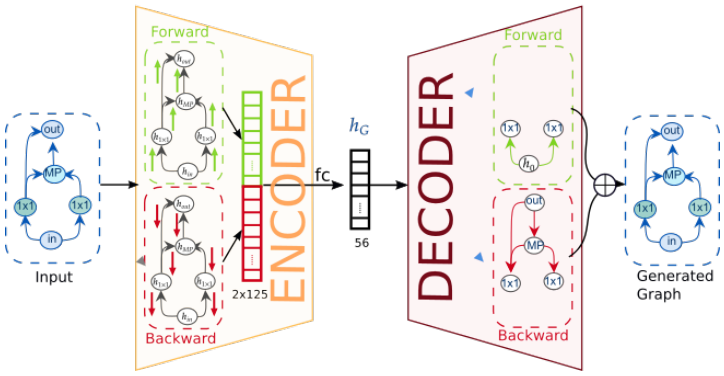


Figure 4.1: Architecture of the proposed SVGe model. It takes as input a neural architecture graph. **(left)** The encoder uses two GNN modules, the forward encoder (green) and the backward encoder (red), to create an informative latent representation $\mathbf{h}_G$. **(right)** This latent vector is input to the decoder, which decodes forward (green) and backward (red) directions separately, generating two graphs in a sequential manner. Their union is the output of SVGe.

- We introduce a novel graph variational autoencoder, SVGe, that builds a structurally smooth variational graph embedding by learning accurate representations of neural architectures.
- We discuss empirical properties of our approach in terms of isomorphic networks.
- We conduct extensive evaluations on the ENAS (Pham et al., 2018), NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201(Dong and Yang, 2020) search spaces and show that our approach allows for competitive BO results in all three search spaces.
- We show that SVGe is able to extrapolate to larger unseen architectures. It finds an architecture with a best accuracy of 95.18% when learning from the NAS-Bench-101 search space. This improves over the best architecture within this space.
- Our top-1 found architecture improves over comparable architectures in terms of validation and test accuracy when transferring to ImageNet16-120 (Chrabaszcz et al., 2017).

The remaining chapter is structured as follows: We briefly review related work for graph generative models in Section 4.1. In Section 4.2, we describe the proposed two-sided GNN variational autoencoders. Section 4.3 discusses empirical properties of our proposed model about the impact of isomorphic networks. The experiments are presented in Section 4.4. Lastly, we conclude this chapter in Section 4.5.

This chapter is a slight modification of the paper J. Lukasik et al. (2021). “Smooth Variational Graph Embeddings for Efficient Neural Architecture Search”. In: *International Joint Conference on Neural Networks (IJCNN)*. The code for this chapter is available on GitHub[1].

4.1 RELATED WORK

Recent years have shown immense progress in representation learning for graph-based data with Graph Neural Networks (GNNs) (Li et al., 2016; Kipf and Welling, 2017; Niepert et al., 2016;

[1] https://github.com/jovitalukasik/SVGe

Hamilton et al., 2017). As introduced in Section 2.2, GNNs follow a message passing scheme, where node feature vectors aggregate information from their neighbors (Gilmer et al., 2017) and capture local structural information. These feature vectors are pooled to obtain a graph-level representation (Ying et al., 2018). GNNs differ in their neighborhood node information as well as in their graph-level aggregation procedure (Scarselli et al., 2009; Hamilton et al., 2017; Kipf and Welling, 2017; Li et al., 2016; Velickovic et al., 2018; Xu et al., 2018; Xu et al., 2019).

Graph generation can be addressed globally by relaxing the adjacency matrix (Kipf and Welling, 2016; Simonovsky and Komodakis, 2018) or sequentially by adding nodes and edges alternately using recurrent networks (Luo et al., 2018; You et al., 2018) or GNNs (Li et al., 2018b; Rezaei et al., 2021). Most graph generation models, especially in NAS, employ variational autoencoders (VAE) (Section 2.3.2). Luo et al. (2018) use an LSTM-based VAE, coupled with performance prediction for gradient-based architecture optimization within the fixed latent space. Different from the previous methods, Huang and Chu (2021) combine a generator with a supernet and search for neural architectures for different device information. Recently, Rezaei et al. (2021) facilitate GNNs in a GAN (Goodfellow et al., 2014) setting using reinforcement learning.

In the aggregation procedure, our decoder model is similar to Li et al. (2018b). Yet, while Li et al. (2018b) treat forward and backward directions equally, our model aggregates node information for both separately to account for the order of network operations and the information flow. Zhang et al. (2019) propose a less efficient, alternating message passing scheme for this purpose and reinstall the validity of decoded architectures using a heuristic which employs prior knowledge on the search space. The proposed method differs in both the encoder and decoder. Our encoder employs an efficient bi-directional model, and the proposed bi-directional decoding facilitates highly accurate reconstructions without constraining the search space.

4.2 Structural Smooth Graph Autoencoding

We aim to learn a structurally smooth latent representation of neural network architectures, which we cast as directed acyclic graphs (DAGs) with nodes representing operations (like convolution or pooling) and edges representing information flow. This enables to (1) accurately predict the accuracy of an unseen graph from training samples and (2) draw new samples which are structurally similar to previously seen ones. Our model SVGe is a variational autoencoder (Kingma and Welling, 2014), as presented in Section 2.3.2.

Below, we provide details on the proposed GNN encoder and decoder models. For NAS, we have to pay particular attention to isomorphic graphs. As they are functionally identical yet represented via distinct adjacency matrices, it is not obvious to guarantee a correct mapping and unique decoding. Motivated by Xu et al. (2019) (see Section 4.3), we chose our model to allow for injective encoding and unique decoding.

4.2.1 *Encoder*

Here, we describe the encoder of our GNN-based SVGe. As already introduced in Section 2.2.1, we denote $G = (V, E)$ as a directed acyclic graph, with nodes $v \in V$ and edges $e \in E$. Each node v has an initial node feature embedding $\mathbf{h}_v^{(0)} \in \mathbb{R}^{d_n}$.

Since our objective is to learn a structurally smooth graph representation, we need to capture the structure and the information flows of the graphs. Thus, our model consists of two encoding

modules, where the messages are passed in the direction of the network's forward pass in the forward encoder (green) and in the direction of the back-propagation in the backward encoder (red) visualized in Figure 4.1 (left). Our variant of GNN formulates the aggregation function $\mathcal{A}(\cdot)$ (Equation (2.2)) as the sum of node message passing modules and uses a single gated recurrent unit (GRU) (Cho et al., 2014) as the update function $\mathcal{U}(\cdot)$ (Equation (2.3)) for both forward and backward encoder.

The forward message passing module $\overrightarrow{f}\left(\overrightarrow{\mathbf{h}}_{\mathrm{u}}^{(k)}, \overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(k)}\right)$ computes a message vector from node u to node v in the k-th iteration, while $\overleftarrow{f}\left(\overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(k)}, \overleftarrow{\mathbf{h}}_{\mathrm{u}}^{(k)}\right)$ is the backward message passing module from node v to u, with $\mathbf{h}_{\mathrm{v}}^{(k)}$ being a feature vector representation of node v at iteration k.

Each graph information direction is aggregated individually:

$$\begin{aligned}\overrightarrow{\mathbf{a}}_{\mathrm{v}}{}^{(k)} &= \sum_{\mathrm{u}\in\mathcal{V}(\mathrm{v})} \overrightarrow{f}\left(\overrightarrow{\mathbf{h}}_{\mathrm{u}}^{(k-1)}, \overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(k-1)}\right)\\ \overleftarrow{\mathbf{a}}_{\mathrm{v}}{}^{(k)} &= \sum_{\mathrm{u}\in\mathcal{V}^{(\mathrm{out})}(\mathrm{v})} \overleftarrow{f}\left(\overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(k-1)}, \overleftarrow{\mathbf{h}}_{\mathrm{u}}^{(k-1)}\right).\end{aligned} \tag{4.1}$$

Recall, $\mathcal{V}(\mathrm{v}) = \{\mathrm{u} \in V \mid (\mathrm{u},\mathrm{v}) \in E\}$ is the set of adjacent nodes to v in the DAG, specifying the network input to v during inference and $\mathcal{V}^{(\mathrm{out})}(\mathrm{v}) = \{\mathrm{u} \in V \mid (\mathrm{v},\mathrm{u}) \in E\}$ are the adjacent nodes to v in the networks backward pass, i.e., in the DAG with reversed edges. Here we also do not use one-hot encoded node labels, but employ a learnable embedding table $\mathrm{L_e}$ on the node types, which stores embeddings (feature vectors) for our initial node embeddings $\overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(0)}$, $\overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(0)}$, as utilized in Section 3.1.1. The functions $\overrightarrow{f}$ and $\overleftarrow{f}$ are implemented using a single fully connected layer (fc). We update the forward and backward node modules using update functions separately in this chapter:

$$\begin{aligned}\overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(k)} &= \overrightarrow{\mathcal{U}}^{(k)}\left(\overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(k-1)}, \overrightarrow{\mathbf{a}}_{\mathrm{v}}{}^{(k)}\right)\\ \overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(k)} &= \overleftarrow{\mathcal{U}}^{(k)}\left(\overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(k-1)}, \overleftarrow{\mathbf{a}}_{\mathrm{v}}{}^{(k)}\right).\end{aligned} \tag{4.2}$$

After the final iteration K, we combine the forward and backward node embeddings $\left(\overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(K)}\right)_{\mathrm{v}\in\mathcal{V}}$, $\left(\overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(K)}\right)_{\mathrm{v}\in\mathcal{V}}$ of node v by concatenation:

$$\left(\mathbf{h}_{\mathrm{v}}^{(K)}\right)_{\mathrm{v}\in V} = \left(\mathrm{CONCAT}\left(\overrightarrow{\mathbf{h}}_{\mathrm{v}}^{(K)}, \overleftarrow{\mathbf{h}}_{\mathrm{v}}^{(K)}\right)\right)_{\mathrm{v}\in V}. \tag{4.3}$$

By combining these two information sets, we capture not only the topology but also the information paths in the graph.

This formulation of aggregation is different to the one in Chapter 3. In the previous chapter, the aggregation module concatenates the forward and backward messages directly at each node and updates this combined information using $\mathcal{U}(\cdot)$ from Equation (2.3), whereas here we consider the forward and backward direction individually for the message passing module. This is important since we are able to consider the information flows separately, which is not the case in the previous chapter.

From the node representations, we compute a graph representation using a gated sum as in Equation (3.4):

$$\mathbf{h}_G = \sum_{v\in V} \mathrm{fc}\left(\mathbf{h}_v^{(K)}\right) \odot \sigma\left(\mathrm{fc}\left(\mathbf{h}_v^{(K)}\right)\right), \tag{4.4}$$

Algorithm 3: Two-Sided Graph Generation

Input: (i) embedding $\mathbf{z} \sim q_\phi(\mathbf{z}|G)$ of graph $G = (V, E)$
Input: (ii) L_d embedding table for node types
Output: reconstructed graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$

1 $\mathbf{h}_{\overrightarrow{G}}, \mathbf{h}_{\overleftarrow{G}} \leftarrow \mathbf{z}$
2 $\overrightarrow{V}, \overrightarrow{E} \leftarrow \text{decodeDirectional}(\text{v}_0, \text{InputType}, \mathbf{h}_{\overrightarrow{G}}, @\overrightarrow{f}_{\text{addNodes}}, @\overrightarrow{f}_{\text{addEdges}})$
$\overleftarrow{V}, \overleftarrow{E} \leftarrow \text{decodeDirectional}(\text{v}_\text{T}, \text{OutputType}, \mathbf{h}_{\overleftarrow{G}}, @\overleftarrow{f}_{\text{addNodes}}, @\overleftarrow{f}_{\text{addEdges}})$
3 $\widetilde{G} = \{\overleftarrow{V} \cup \overrightarrow{V}, \overleftarrow{E} \cup \overrightarrow{E}\}$

with a sigmoid activated fc layer $\sigma(\text{fc}(\cdot))$ as a gating function, a linear activated fc layer, and $\odot$ denoting the Hadamard product. Since we use this encoder in a variational autoencoder setting, we add an extra graph aggregation layer equal to Equation (4.4) to obtain $\mathbf{h}_G^{\text{var}}$. Thus, the outputs of our encoder are the parameters of the approximate posterior distribution $q_\phi(\mathbf{z}|G) = \mathcal{N}(\mathbf{h}_G, \mathbf{\Sigma})$, with $\mathbf{h}_G$ being the mean and $\mathbf{h}_G^{\text{var}}$ the diagonal of the variance-covariance matrix $\mathbf{\Sigma}$ of the multivariate normal distribution. Section 4.3 discusses the properties of this encoder w.r.t. injectivity and isomorphic graphs.

4.2.2 *Decoder*

The SVGe decoder $p_\theta(G|\mathbf{z})$ takes a latent point $\mathbf{z}$ as input and reconstructs G simultaneously from two directions (start-to-end $(\overrightarrow{V}, \overrightarrow{E})$ and end-to-start node $(\overleftarrow{V}, \overleftarrow{E})$), see Figure 4.1 (right). As in the encoder, the model explicitly learns a neural architecture's forward *and* backward pass, allowing for highly accurate reconstructions of graphs without "loose ends". An overview of the general graph generation process using the two directional graph generations processes is given in Algorithm 3.

The directional graph generation starts from the input node v_0 for forward decoding and the output node v_T for backward decoding. Each graph is built iteratively in a sequence of operations that add nodes and edges until the end node is generated, similar to Zhang et al. (2019).

The union of both, forward and backward graph, builds the output graph. We will describe the directional decoding in more detail in the following.

Directional Decoding.

The directional graph generation starts from an initial node with type "InputType" for the forward decoding and with type "OutputType" for the backward decoding. The input node and all generated nodes v_t are embedded according to the *initNode* module:

$$\mathbf{h}_t = f_{\text{initNode}}\big(\mathbf{z}, \mathbf{h}_{\widetilde{G}^{(t)}}, \text{L}_\text{d}[\text{type}]\big), \tag{4.5}$$

with f_{initNode} being a two-layer fc with ReLU activation. It takes as input the sampled point $\mathbf{z}$, the partial graph embedding $\mathbf{h}_{\widetilde{G}^{(t)}}$ and the learned node type embedding $\text{L}_\text{d}[\text{type}]$, If the node v_t is either the input or the output node, Equation (4.5) simplifies to $\mathbf{h}_t = f_{\text{initNode}}(\mathbf{z}, \text{L}_\text{d}[\text{type}])$.

Algorithm 4: decodeDirectional

Input: (i) start node v_0 with $\text{type}_{v_0} \in \{\text{InputType}, \text{OutputType}\}$
Input: (ii) graph embedding $\mathbf{h}_G$
Input: (iii) $@f_{\text{addNodes}}, @f_{\text{addEdges}}$
Output: node set V and edge set E

$V \leftarrow \{v_0\}, E \leftarrow \emptyset, t = 0$
initialize InputNode v_0, with embedding $\mathbf{h}_0 \leftarrow f_{\text{initNode}}(\mathbf{z}, L_d[\text{type}_{v_0}])$; ▷ Eq. (4.5)
while $\text{type}(v_t) \neq \text{EndNode}$ **do**
  $V \leftarrow V \cup \{v_{t+1}\}$; ▷ add node
  $s_{\text{addNode}} \leftarrow f_{\text{addNode}}(\mathbf{z}, \mathbf{h}_G)$; ▷ Eq. (4.7)
  $\text{type}(v_{t+1}) \sim \text{Categorical}(s_{\text{addNode}})$; ▷ get type Eq. (4.6)
  $\mathbf{h}_{t+1} \leftarrow f_{\text{initNode}}(\mathbf{z}, \mathbf{h}_G, L_d[\text{type}(v_{t+1})])$; ▷ Eq. (4.5)
  for $v_j \in V \setminus v_{t+1}$ **do**
    $e_{(j,t+1)} \sim \text{Ber}(f_{\text{addEdges}}(\mathbf{h}_{t+1}, \mathbf{h}_t, \mathbf{h}_G, \mathbf{z}))$; ▷ sample whether to add edge, Eq. (4.8)
    if $e_{(j,t+1)} = 1$ **then**
      $E \leftarrow E \cup \{e_{(j,t+1)} = (v_j, v_{t+1})\}$; ▷ add edge
    end
  end
  $\mathbf{h}_{\widetilde{V}} \leftarrow \text{concat}(\mathbf{h}_t, \mathbf{h}_{t+1})$
  $\mathbf{h} \leftarrow (\mathbf{h}_{\widetilde{V},G})$; ▷ update node embeddings Eq. (4.1), (4.2)
  $\mathbf{h}_G \leftarrow \text{aggregate}(\mathbf{h})$; ▷ update graph embedding Eq. (4.3), (4.4)
  $t \leftarrow t + 1$
end

Given the start node v_0, its embedding $\mathbf{h}_0$ and the graph embedding $\mathbf{h}_G = \mathbf{z}$, a graph is generated by iterating over a sequence of modules whose weights are shared across iterations until an end node, e.g., of type OutputNode in the forward decoder, is drawn.

In every iteration, a new node is created and added to the graph and its node type (i.e., operation in the network architecture) is selected by the *addNode* module. It takes as input the representation of the partial graph $\mathbf{h}_{\widetilde{G}^{(t)}}$ and the sampled point $\mathbf{z}$ and determines the next missing node

$$\text{NodeType} \sim \text{Categorical}\left(s_{\text{addNode}}^{(t+1)}\right), \tag{4.6}$$

where

$$s_{\text{addNode}}^{(t+1)} = f_{\text{addNode}}\left(\mathbf{z}, \mathbf{h}_{\widetilde{G}^{(t)}}\right). \tag{4.7}$$

f_{addNode} are two fc layers with ReLU-non-linearities, producing parameters for the categorical node type distribution from which the next node is sampled. This newly added node is then initialized with the *initNode* module (Equation (4.5)) and added to the already existing propagated node embedding, $\mathbf{h}_{\widetilde{V}} = \text{CONCAT}((\mathbf{h}_j)_{0 \leq j \leq t}, \mathbf{h}_{t+1})$.

To every added node v_{t+1}, the decoder creates edges from already existing nodes according to a scoring function, where a high value represents a likely edge. This *addEdges* module takes as input the partial graph node embeddings $\mathbf{h}_{t+1}$ and $\mathbf{h}_t = (\mathbf{h}_j)_{0 \leq j \leq t}$, as well as the partial graph

embedding $\mathbf{h}_{\widetilde{G}^{(t)}}$ and the sampled point $\mathbf{z}$, leading to

$$e_{(i,t+1)} \sim \text{Ber}\big(f_{\text{addEdges}}\big(\mathbf{h}_{t+1}, \mathbf{h}_t, \mathbf{h}_{\widetilde{G}^{(t)}}, \mathbf{z}\big)\big), \tag{4.8}$$

where Ber denotes a Bernoulli distribution. Sampling from this distribution yields the new set of directed edges ending in v_{t+1}. f_{addEdges} is again a two-layer fc layer with ReLU activation.

After adding the edges, the concatenated node embeddings $\mathbf{h}_{\widetilde{V}}$ are aggregated and updated in the *prop* module, similar to the encoder using Equation (4.1) and Equation (4.2), yielding the updated node embedding $\mathbf{h}$. These node embeddings are aggregated into a single graph representation $\mathbf{h}_G$ according to Equation (4.4). An overview of the directional decoding is given in Algorithm 4. Note, the encoder and decoder GNNs have distinct weights.

4.2.3 *Loss Function and Training*

We train the encoder and decoder of SVGe jointly in an unsupervised manner. Given a fixed node ordering of the DAG, which we discuss in Section 4.3, we know the ground truth of the outputs of *AddNode* (Equation (4.6)) and *AddEdges* (Equation (4.8)) during training. We use this ground truth to compute a node-level loss $\mathcal{L}_V^t$, which is a Cross-Entropy loss, and an edge-level loss $\mathcal{L}_E^t$, which is a Binary Cross-Entropy loss, at each iteration t. Additionally, we replace the model output by the ground truth such that possible errors will not accumulate throughout iterations.This is also known as teacher forcing (Williams and Zipser, 1989). To compute the overall reconstruction loss for a graph G, we sum up node losses and edge losses over all iterations for both decoding directions; $\mathcal{L}_{\text{rec}} = \overrightarrow{\mathcal{L}_V} + \overrightarrow{\mathcal{L}_E} + \overleftarrow{\mathcal{L}_V} + \overleftarrow{\mathcal{L}_E}$. Following Kingma and Welling (2014), we assume $p_\theta(\mathbf{z}) \sim \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbb{1})$ and $p_\theta(G|\mathbf{z}) \sim \mathcal{N}(\mathbf{h}_G, \boldsymbol{\Sigma})$. Furthermore, we approximate the posterior by a multivariate Gaussian distribution with diagonal covariance structure. This can be written as $\log q_\phi(\mathbf{z}|G) = \log \mathcal{N}(\mathbf{z}; \mathbf{h}_G, \boldsymbol{\Sigma})$ and ensures a closed form of the KL divergence, as presented in Section 2.3.2:

$$\text{D}_{\text{KL}} = -\frac{1}{2}\sum_{j=1}^{J}\big(1 + \log\big(\mathbf{h}_G^{\text{var}}\big)_j - (\mathbf{h}_G)_j^2 - (\mathbf{h}_G^{\text{var}})_j\big). \tag{4.9}$$

Thus, the overall variational autoencoder loss from Equation (2.24) becomes:

$$\mathcal{L} = \mathcal{L}_{\text{rec}} + \alpha \text{D}_{\text{KL}}, \tag{4.10}$$

with α additionally regularizing the Kullback-Leibler divergence. Following Jin et al. (2018) and Zhang et al. (2019), we set $\alpha = 0.005$.

Furthermore, for training SVGe jointly with the surrogate model for performance prediction, we modify the loss Equation (4.10) by including the performance prediction loss $\mathcal{L}_{\text{pred}}$:

$$\mathcal{L} = \beta(\mathcal{L}_{\text{rec}} + \alpha \text{D}_{\text{KL}}) + (1 - \beta)\mathcal{L}_{\text{pred}}. \tag{4.11}$$

4.3 Discussion of the Impact of Isomorphisms

In the following, we discuss SVGe in the context of isomorphic input graphs. Intuitively, if isomorphic graphs, i.e., graph representations of the same neural architecture, are mapped to distinct latent points, the latent space is intrinsically redundant. This hampers an efficient embedding of structural similarity. Conversely, if non-isomorphic graphs are mapped to the same latent point,

their performance can neither be correctly predicted nor their structure reconstructed. Thus, a suitable graph encoder has to map any two isomorphic graphs to the same latent point (*ISO*). A suitable decoder decodes each latent point to a unique graph and preserves the difference between non-isomorphic graphs (*INJ*).

4.3.1 *Unique Latent Space Representation.*

Here, we discuss the proposed SVGe *encoder* w.r.t. properties (*ISO*) and (*INJ*).

Theorem 3 in Xu et al. (2019) gives sufficient conditions on injectivity (*INJ*) of the GNN's node aggregation module and its update module. However, the required existence of an appropriate injective aggregation function operating on multisets can only be guaranteed theoretically on countable input feature spaces. Even then, Xu et al. (2019) give no explicit construction but argue that it can be approximated via MLPs. Thus, we verify empirically that SVGe maps isomorphic graphs to the same embedding (*ISO*) in an experiment on 11 606 isomorphic graph pairs of length 7 from the NAS-Bench-101 search space. 100% of such isomorphic pairs were mapped onto the same point. The mapping of distinct graphs to distinct latent representations (*INJ*) is a prerequisite for accurate reconstruction and therefore validated in the reconstruction ability in Section 4.4.1.

4.3.2 *Decoding from the Latent Space.*

We now discuss how the *decoder* handles isomorphic DAGs w.r.t. (*INJ*). Since isomorphic graphs are mapped onto the same latent point by the encoder (see above), it suffices for the decoder to decode them uniquely, i.e., deterministically. It can be easily seen that this is the case for SVGe.

Yet, how can the decoder be trained efficiently to decode one out of several isomorphic graphs from the same latent point, when decoding graph G_1 instead of an isomorphic graph G_2 leads to significant reconstruction loss? Isomorphic graphs can be created from one another by permuting nodes in the adjacency matrix. Thus, to ease the decoder into the learning process we bring the graphs in a unified form. Towards a unique representation, we limit the training data to upper triangular matrices.

The remaining isomorphic graphs are removed from the training set as in Ying et al. (2019), since we need a unique representation for each graph. Note, this only removes duplicate architectures. For the ENAS and the NAS-Bench-201 search space, the adjacency matrices are unique, given the upper triangular representation.

Given such clean training data, the choice of the VAE decoder is still crucial. For good reconstruction from the latent space, we introduce a two-sided decoder, which captures the information flow from the input to the output node and vice versa. Specifically, nodes generated in the forward direction that are not connected to the output are connected to their predecessor in the backward decoder, and vice versa. The union of both forward and backward decoded graphs will therefore likely contain all missing edges from both single decoders. D-VAE (Zhang et al., 2019) overcomes this problem of possible trailing nodes by incorporating a heuristic "post-processing" step employing prior knowledge on the search space: It connects each non-output node with out-degree zero to the output node.

Search Space	Method	Accuracy ↑	Validity ↑	Uniqueness ↑	Novelty ↑
ENAS	D-VAE (Zhang et al., 2019)	**99.96**	**100.00**	37.26	**100.00**
	DGMG (Li et al., 2018b)	99.29	**100.00**	37.55	**100.00**
	SVGe (ours)	99.63	**100.00**	**39.03**	**100.00**
NAS-Bench-101	D-VAE	25.89	82.55	19.84	16.52
	arch2vec (Yan et al., 2020)	98.84	43.70	10.00	**82.84**
	DGMG	**99.99**	**89.70**	29.24	16.72
	SVGe (ours)	99.57	79.16	**32.10**	16.37
NAS-Bench-201	arch2vec	**99.99**	95.93	7.33	**13.26**
	DGMG	99.97	**100.00**	5.35	12.62
	SVGe (ours)	**99.99**	**100.00**	**8.28**	10.24

Table 4.1: VAE abilities on ENAS, NAS-Bench-101, and NAS-Bench-201 in %.

4.4 Experiments

We evaluate the proposed SVGe on three different, commonly used search spaces from the NAS literature: the two tabular benchmarks NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201 (Dong and Yang, 2020), as well as architectures from the ENAS (Pham et al., 2018) search space.

ENAS Search Space The ENAS (Pham et al., 2018) search space consists of architectures represented by a DAG with $|V| = 8$ nodes (including the input and output node) and 6 operation choices on each of the non-input and non-output nodes, $O = \{$ 3× 3 conv., 5 × 5 conv, 3× 3 depthwise-conv, 5× 5 depthwise.conv., 3× 3 max pooling, 3× 3 avg. pooling $\}$. We use the same 19 020 searched architectures as Zhang et al. (2019) from this space.

While the NAS-Bench-101 benchmark provides the true performance of all fully trained architectures, ENAS does not provide any such value. Therefore, we use the weights of the optimized one-shot model as a proxy for the validation/test performance of the sampled architectures during optimization.

We split the (architecture, accuracy) pairs into 90% training and 10% test examples for each search space to train our autoencoder model. After testing its autoencoder abilities, we evaluate SVGe on performance prediction, BO and search space extrapolation. In addition, we show the transferability of our BO and extrapolation results.

All the algorithms and routines are implemented using PyTorch (Paszke et al., 2019) and PyTorch Geometric (Fey and Lenssen, 2019). We provide additional information about the used hyperparameters for this chapter in Appendix B.2 and representations of the NAS-Bench-101 and NAS-Bench-201 search spaces in Appendix A.1, Appendix A.2.

4.4.1 *Autoencoder Abilities.*

Following previous work (Zhang et al., 2019; Jin et al., 2018), we evaluate SVGe by means of *reconstruction accuracy*, *validity*, *uniqueness* and *novelty*.

We evaluate these abilities on the ENAS, NAS-Bench-101 and NAS-Bench-201 search spaces and compare to Zhang et al. (2019) and Yan et al. (2020). As an ablation on ENAS, NAS-Bench-101 and

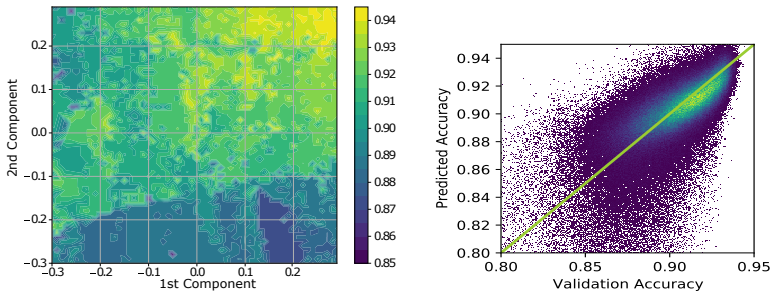


Figure 4.2: **(left)** Visualization of the first two principal components of the latent space for SVGe using NAS-Bench-101 architectures with a validation accuracy above 75% on CIFAR-10. **(right)** Performance prediction of the fine-tuned SVGe on the NAS-Bench-101 test data with a validation accuracy above 80% on CIFAR-10.

NAS-Bench-201, we also adapted the generative model from Li et al. (2018b), *DGMG* in conjunction with our encoder architecture. We train the models on 90% of the dataset and test it on the 10% held-out data.

Table 4.1 shows the results. D-VAE (Zhang et al., 2019) and all our model variants show reasonable performance w.r.t. the *reconstruction accuracy*, on ENAS. On NAS-Bench-101 and NAS-Bench-201, our approaches perform equally well and are comparable to arch2vec (Yan et al., 2020), while D-VAE performs poorly on NAS-Bench-101 and diverges on NAS-Bench-201. We hypothesize that D-VAE's hard constraints on the graph decoding are not suitable for NAS-Bench-101 and NAS-Bench-201. The resulting latent space cannot capture all relevant information, without introducing more model modifications.

The *validity* measures how many of the decoded samples are valid DAGs. Note, the validity is defined in each search space individually by its topology and network constraints. Again, we see good overall performance on ENAS. On NAS-Bench-101 SVGe, DGMG and D-VAE perform comparably, while the validity for arch2vec is low, indicating that their decoder is not suited for graph generation. This trend is less severe yet observable on NAS-Bench-201.

The *uniqueness* ability measures the unique share of the valid decoded graphs. This measure is particularly important for any kind of extrapolation from the training data. If the uniqueness is small, distinct and potentially distant latent points are decoded to the same output. This may hurt subsequent approaches, such as Bayesian optimization.While all approaches could be improved w.r.t. uniqueness, SVGe performs better than previous models.

The *novelty* indicates the portion of graphs from the valid graph set which have not been observed during training. Here, values on ENAS are high and non-informative since only a small portion of the search space is covered by training data. On the two NAS-Bench variants, D-VAE and our models perform on par while the numbers for arch2vec are higher. The direct comparison of these values is impaired since arch2vec issues an overall lower number of valid graphs. We conclude that SVGe shows good behaviour in terms of accuracy, validity and uniqueness over all three search spaces.

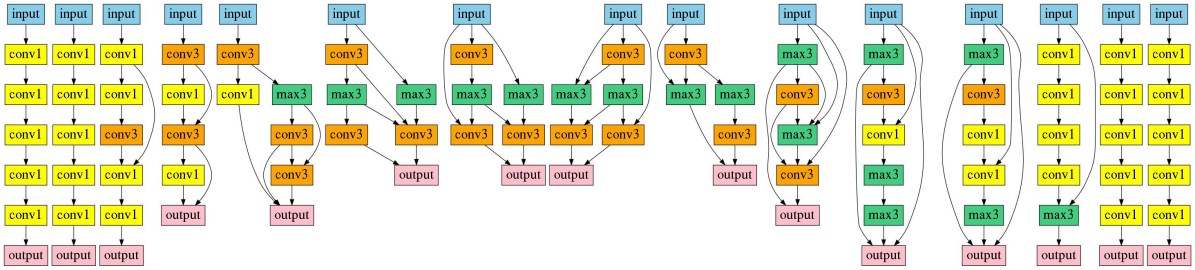


Figure 4.3: Circle in the latent space of NAS-Bench-101 with 14 equidistant sampled points on a hypersphere. The graphs vary smoothly with increasing distance from the start.

Surrogate Model	**Performance Prediction**(MSE in 10^{-3}) ↓		
	1 000	10 000	100 000
Semi-Supervised Assessor (Tang et al., 2020)	$3.1 \pm 3 \times 10^{-4}$	$2.6 \pm 2 \times 10^{-4}$	$1.6 \pm 2 \times 10^{-4}$
D-VAE (Zhang et al., 2019)	$3.9 \pm 3 \times 10^{-4}$	$2.6 \pm 2 \times 10^{-4}$	$2.0 \pm 9 \times 10^{-5}$
DGMG (Li et al., 2018b)	$3.7 \pm 1 \times 10^{-4}$	$2.7 \pm 3 \times 10^{-6}$	$2.0 \pm 1 \times 10^{-4}$
GNN Encoder (Chapter 3)	$4.4 \pm 3 \times 10^{-4}$	$\mathbf{2.2} \pm 1 \times 10^{-4}$	$\mathbf{1.5} \pm 1 \times 10^{-5}$
SVGe (ours)	$\mathbf{2.8} \pm 2 \times 10^{-5}$	$2.3 \pm 4 \times 10^{-5}$	$2.0 \pm 3 \times 10^{-5}$

Table 4.2: Comparison of MSE and standard deviation for performance prediction on NAS-Bench-101.

4.4.2 *Latent Space Smoothness Observations*

In Figure 4.2 and Figure 4.3, we visualize the smoothness of the SVGe graph embedding in the NAS-Bench-101 search space. Figure 4.2 (left) visualizes the SVGe embedding. We plot equidistant points within a $[-0.3, 0.3]$ grid, given a 2D subspace of our training data with a validation accuracy above 75% spanned by the first two principal components. Architectures with similar accuracies are close to each other and high accuracy architectures form clusters. Figure 4.3 shows a unit circle in a randomly chosen orthogonal direction in the SVGe embedding space. We start from a flat net encoding in the latent space and randomly pick 14 equidistant datapoints along the hypersphere returning to the start point. These datapoints are decoded and visualized as architectures. As one can see they change smoothly with changing only few operations and edges at each step.

4.4.3 *Performance Prediction from Latent Space*

Next, we evaluate SVGe in terms of performance prediction on NAS-Bench-101 architectures. This allows for direct comparison to the recent work (Tang et al., 2020). We train SVGe on 90% of all 423 624 datapoints in NAS-Bench-101 for reconstruction to obtain the latent space. Then, we fine-tune the unsupervisedly trained model for performance prediction using a regression model, which is a four-layer MLP with ReLU non-linearities. The SVGe model and the regression model are trained jointly for performance prediction on 1 000/10 000/100 000 randomly sampled architectures with test accuracies from NAS-Bench-101. For comparison, we also train D-VAE and DGMG in the same setting. We compare the ability to predict performances accurately on the validation set (10% of the overall (architecture, accuracy) pairs). Table 4.2 shows the MSE, which denotes the empirical squared loss between the predicted and ground truth data, and the

Search Space	**Method**	**Val Acc.**(%) ↑	**Test Acc.**(%)↑	**VAE Training** (in GPU days)
ENAS	D-VAE (Zhang et al., 2019)	-	94.80	0.7
	SVGe (ours)	-	**95.11**	**0.21**
NAS-Bench-101	*oracle*	95.15	94.09	-
	DGMG (Li et al., 2018b)	94.08	93.51	1.65
	SVGe (ours)	94.60	93.88	2.01

Table 4.3: Bayesian optimization on the ENAS and NAS-Bench-101 search spaces. SVGe slightly outperforms D-VAE on ENAS and reduces the runtime (in GPU hours) by a factor of 3.

standard deviation of 3 runs.

Our proposed SVGe has a slightly lower MSE compared to Tang et al. (2020), which focus precisely on this subproblem, when few annotated datapoints are given. This is important in particular for NAS, since every training sample corresponds to a fully evaluated architecture and is thus expensive. D-VAE and DGMG show high MSEs for this small amount of training data. We expected this behavior for D-VAE because it already showed poor abilities in Section 4.4.1 on NAS-Bench-101. The low prediction accuracy for DGMG hints to potential overfitting in the autoencoder. In Figure 4.2 (right), we plot the performance prediction ability of our model trained on 1 000 sampled architectures from Table 4.2 for high-performing architectures (above 80% test accuracy). This figure shows a strong correlation between predicted and true accuracies.

4.4.4 *Bayesian Optimization*

We have seen in the previous experiments that the proposed SVGe generates a latent space which enables to interpolate from seen labels/performances and outperforms D-VAE and DGMG significantly. Next, we perform NAS via BO in the ENAS search space, in order to allow a fair comparison to D-VAE (Zhang et al., 2019) by exchanging only the D-VAE generative model with our SVGe and using exactly the same setup as in Zhang et al. (2019).

We perform 10 iterations of batch BO (with a batch size of 50). See Section 2.3.1 for a general overview of Bayesian optimization. We average the results across 10 trials based on a sparse Gaussian process (SGP) (Snelson and Ghahramani, 2005) with 500 inducing points and expected improvement (Equation (2.15)) (Mockus, 1974) as acquisition function. Following Kusner et al. (2017) and Zhang et al. (2019), we use the acquisition function to select a new batch of points in each iteration and assume the target label for each new point in the batch to be the mean of the predictive distribution of the SGP (Section 2.3.1). Each point in this selected batch is then decoded to its discrete graph structure using our decoder and evaluated using either the available ground truth accuracy as in NAS-Bench-101 (Ying et al., 2019) or the weight-sharing accuracy for ENAS (Pham et al., 2018). These (architecture, accuracy) pairs are then added to the training data, which is used for the SGP for the next iteration. In the case of an invalid generated DAG, we are not able to obtain the ground truth accuracy and therefore set the accuracy equal to the worst performance in the training data.

For the evaluation of BO in ENAS, we select the best found 15 architectures w.r.t. their weight-sharing accuracies and fully train them from scratch on CIFAR-10. As shown in Table 4.3, SVGe's best found architecture achieves an accuracy of 95.11%, which is 0.31 percent points better than the best found architecture using the D-VAE embedding. Table 4.3 also reports compute times for

model training on ENAS. It shows that SVGe can be trained more efficiently than D-VAE.

Additionally, we perform BO on the NAS-Bench-101 search space with our SVGe model and optimize on validation accuracies. We train the SGP initially on 1 000 randomly sampled architectures in each trial. Because of its low performance prediction on NAS-Bench-101, D-VAE is expected to perform poorly in this setting. To assess our results in Table 4.3, we report as *oracle* the best NAS-Bench-101 architecture in terms of validation accuracy and its test accuracy.

BO on SVGe yields a model with 94.60% validation and 93.88% test accuracy, improving over the best found architecture using DGMG in terms of both validation and test accuracy. When using all our training data (90% of NAS-Bench-101) to train the SGP, SVGe's best found architecture achieves a validation accuracy of 94.67%. This architecture yields a test accuracy of 94.26% on NAS-Bench-101 which is higher than the test accuracy of the best NAS-Bench-101 architecture in terms of validation accuracy. Note that the best *oracle* test accuracy would be 94.45% (at only 94.87% validation accuracy).

Since the D-VAE training diverges on the NAS-Bench-201 search space, we can not conduct a direct comparison. Yan et al. (2020) perform BO in their latent space, using DNGO (Snoek et al., 2015) instead of SGP, and define the current state-of-the-art. DNGO uses a Bayesian linear regression model in conjunction with a deep neural network and overcomes the cubical scales of a GP. This problem is also already mentioned in Section 2.3.1. DNGO scales linearly with the number of training data and cubically with the input dimension, and is therefore suited for low-dimensional embedding spaces while it performs less well on high dimensional spaces as ours. Conversely, using SGP on low-dimensional embedding spaces is sub-optimal. Therefore, a direct comparison to arch2vec in terms of BO should be taken with caution. Performing BO in the SVGe generated latent space yields a test accuracy of 93.38% on the CIFAR-10 image classification task on NAS-Bench-201. In comparison, arch2vec yields a mean test accuracy of 94.18%, which only leaves a small gap. Thus, SVGe is able to find well-performing architectures in all three search spaces.

4.4.5 *Extrapolation Ability*

Finally, we show that our smooth embedding space enables to find better architectures than the ones mentioned above even without dedicated optimization approaches by simple extrapolation from the labeled dataset. We have already seen the ability for extrapolation of the GNN encoder in Chapter 3. Therefore, we employ the ability of SVGe to predict neural architectures' performances on CIFAR-10 with more nodes and edges than seen at training time in both NAS-Bench-101 and ENAS search spaces here in this chapter.

On the NAS-Bench-101 search space, we generate graphs (cells) containing 8 nodes. Note that our SVGe model has never seen such architectures during training (NAS-Bench-101 is limited to cells with up to 7 nodes). To generate these new graphs, we pick the best performing graph from NAS-Bench-101 based on the validation accuracy and expand it to graphs with 8 nodes, maintaining the upper triangular matrices structure (1 384 graphs in total). From these graphs, we select the top-5 architectures with the highest predicted validation accuracy using SVGe with the surrogate model (see Section 4.4.3, trained on 1 000 graphs). These models are trained from scratch on CIFAR-10 using the training pipeline from Ying et al. (2019). As shown in Table 4.4, the architecture found by extrapolating using our SVGe model achieves a top-1 validation accuracy of 95.18% and a test accuracy of 94.92% for graphs of length 8, which improves over 0.83% in test

Search Space	Method	Val Acc. (%) ↑	Test Acc. (%) ↑
NAS-Bench-101 7	*oracle*	95.15	94.09
NAS-Bench-101 8	SVGe (ours)	**95.18**	**94.92**
ENAS 14	D-VAE+BO (Zhang et al., 2019)	**96.12**	-
	SVGe (ours)	96.09	-

Table 4.4: Architecture extrapolation experiments.

Method	Val Acc.(%) ↑	Test Acc.(%) ↑
NAS-Bench-201 (optimal) (Dong and Yang, 2020)	46.77	47.31
ResNet (Dong and Yang, 2020)	44.53	43.63
SVGe + BO (NB101-7) (ours)	54.70	**56.83**
SVGe + Zero-Shot (NB101-8) (ours)	**55.13**	55.53

Table 4.5: Dataset transfer learning to ImageNet16-120 (Chrabaszcz et al., 2017).

accuracy over the best 7-nodes architecture test accuracy.

On the ENAS search space, we evaluate SVGe on the macro architecture containing a total of 14 nodes (layers, including the input and output node) compared to architectures with 8 nodes used during the SVGe training. See Pham et al. (2018) for more information about the macro architecture with 14 nodes. We further fine-tune the embedding space by sampling 1 000 architectures from the training set and train the SVGe together with the performance predictor. Note that this performance predictor uses the weight-sharing accuracies as proxy for the true accuracy of the fully trained architectures. We select the top-5 architectures based on the predicted validation performance and again fully train them on CIFAR-10, using the settings from Zhang et al. (2019). As shown in Table 4.4, the best found architecture in the ENAS 14 search space achieves a validation accuracy of 96.09% which is close to the one found by D-VAE. Note, D-VAE (Zhang et al., 2019) used a Bayesian optimization approach on this macro architecture search space as in Section 4.4.4 to find this architecture, whereas SVGe can achieve similar results by direct extrapolation (aka zero-shot prediction).

Last, we test the transferabilty to other datasets of the architectures found by our model. For that purpose, we train the best found architecture in the BO experiment and the top-1 architecture found via extrapolation on ImageNet16-120 (Chrabaszcz et al., 2017) in the training scheme from Dong and Yang (2020). As shown in Table 4.5 both architectures improve over a comparably deep ResNet architecture (He et al., 2016) and the best NAS-Bench-201 architecture by a significant margin.

4.5 CONCLUSION

In this chapter, we propose SVGe, a Smooth Variational Graph embedding model for NAS. We give empirical results on SVGe encoding abilities and show that it applies more easily to new search spaces than previous approaches (Zhang et al., 2019). We present results on the NAS-Bench-101, NAS-Bench-201 and ENAS search spaces and show good results for performance prediction surrogate models and Bayesian optimization in the smooth embedding space. Furthermore, we

demonstrate the extrapolation abilities of SVGe to larger unseen graphs to find high-performing architectures. Image dataset transfer experiments to ImageNet16-120 also show that the found high-performing architectures can improve over the performance of comparable architectures by a significant margin.

Generative NAS with Latent Space Optimization

5

In this thesis, we have already mentioned that classical search methods are inefficient, which ultimately led to new more efficient search methods based on surrogate models, or learned architecture representations in conjunction with generative models. These methods aim to further improve the query efficiency search in NAS, which is crucial, since each query implies a full training and evaluation of the neural architecture on the underlying target dataset. In Chapter 3, we presented a GNN surrogate model that is able to correctly predict architecture's performances, especially in the setting of zero-shot prediction. Building on this, we presented in Chapter 4 a generative NAS model that, using a GNN variational autoencoder, spans a smooth latent space that facilitates classical search methods such as BO, but also performance prediction using a surrogate model, which leads to competitive search results. Both presented methods aim improving the query efficiency and finding high-performing architectures.

This trade-off between query efficiency and resulting high-performing architectures is an active research field. Yet, no attempts were made so far to leverage the advantages of both search paradigms. Therefore, we propose a model that incorporates the focus of promising architectures already in the architecture generation process by optimizing the latent space *directly*: We let the generator learn in which areas of the data distribution to look for promising architectures. This way, we reduce the query amount even further, resulting in a query efficient and very effective NAS search method. Our proposed method is inspired by a latent space optimization (LSO) technique (Tripp et al., 2020), originally used in the context of variational autoencoders to optimize generated images or arithmetic expressions using BO. We adapt this concept to NAS and pair it with an architecture performance predictor in an end-to-end learning setting, such that it allows us to iteratively reshape the architecture representation space. Thereby, we promote desired properties of generated architectures in a highly query-efficient way, i.e., by learning expert generators for promising architectures. Since we couple the generation process with a surrogate model to predict desired properties such as high accuracy or low latency of generated architectures, there is no need in our method for BO in the generated latent space, making our method even more efficient.

In practice, we pretrain a GNN-based generator network on a target space of neural architectures, which does not rely on any architecture evaluation and is therefore fast and query-free. The generator is trained in a novel generative setting that directly compares generated architectures to randomly sampled architectures using a reconstruction loss without the need of a discriminator network as in generative adversarial networks (GANs) (Goodfellow et al., 2014) or an encoder as in variational autoencoders (VAEs) (Kingma and Welling, 2014) (see also Section 2.3.2). We use an MLP as a surrogate to rank performances and hardware properties of generated architectures. In contrast, previous generative methods either rely on training and evaluating supernets (Huang and Chu, 2021), which are expensive to train and dataset specific, or pretrain a latent space and search within this space directly using BO (Zhang et al., 2019; Yan et al., 2020), reinforcement learning (Rezaei et al., 2021) or gradient-based methods (Luo et al., 2018). These methods incorporate either GANs, which can be hard to train or VAEs, which are biased by the regularization, whereas our plain generative model is easy to train. In addition we enable backpropagation from the performance predictor to the generator. Thereby, the generator can efficiently learn which

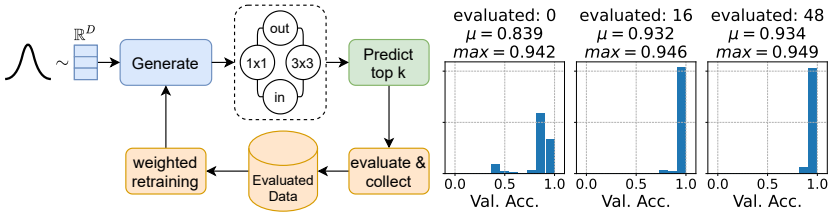


Figure 5.1: **(left)** Our search method generates architectures from points in an architecture representation space that is iteratively optimized. **(right)** The architecture representation space is biased towards better-performing architectures with each search iteration. After only 48 evaluated architectures, our generator produces state-of-the-art performing architectures on NAS-Bench-101.

part of the architecture search space is promising with only few evaluated architectures.

By extensive experiments on common NAS benchmarks (Ying et al., 2019; Dong and Yang, 2020; Zela et al., 2022; Klyuchnikov et al., 2022; Li et al., 2021a) as well as ImageNet (Deng et al., 2009), we show that our method, called *AG-Net*, is effective and sample-efficient. It reinforces the generator network to produce architectures with improving validation accuracy (see Figure 5.1), as well as in improving on hardware-dependent latency constraints (see Figure 5.5) while keeping the number of architecture evaluations small. In summary, we make the following contributions:

- We propose a simple model that learns to focus on promising regions of the architecture space. It can thus learn to generate high-scoring[1] architectures from only a few queries.
- We learn architecture representation spaces via a *novel generative design* that is able to generate architectures stochastically while being trained with a simple reconstruction loss. Unlike VAEs (Kingma and Welling, 2014) or GANs (Goodfellow et al., 2014), neither encoder network nor discriminator network is necessary.
- Our model allows sample-efficient search and achieves state-of-the-art results on several NAS benchmarks as well as on ImageNet. It allows joint optimization w.r.t. hardware properties in a straightforward way.

The remaining chapter is structured as follows: We will provide more information on the latent space optimization in Section 5.1. Section 5.2 describes the proposed model in detail. The experiments are presented in Section 5.3 and additional ablation studies in Section 5.4. We conclude this chapter in Section 5.5.

This chapter is a slightly enhanced version of the paper J. Lukasik et al. (2022). “Learning Where to Look - Generative NAS is Surprisingly Efficient”. In: *Proc. of the European Conference on Computer Vision (ECCV)*. The experiments in Section 5.3.4 have been conducted by Steffen Jung. The code for this chapter is available on GitHub[2].

[1] We use *scoring* here to distinguish that we not only search for high-performing architectures, but also for other objectives, such as low latency. Therefore, scoring is used in this chapter as a general target variable.

[2] https://github.com/jovitalukasik/AG-Net

5.1 Latent Space Optimization using Weighted Retraining

5.1.1 *Problem Statement*

We introduced the variational autoencoder and an optimization example within its generated latent space by means of BO in Section 2.3.2. The latent space generation and optimization steps are combined in a two-step approach, since the optimization step is only used post-hoc on the pretrained latent space. This two-step approach leads therefore to a decoupling of the generation part and the optimization. Tripp et al. (2020) analyze the influence of this decoupling and reveals some shortcomings, which we will discuss in the following.

First, recall that the VAE model is trained with a prior $p_\theta(\mathbf{z})$ and a generative model $p_\theta(\mathbf{x}|\mathbf{z})$ on the objective of the ELBO Equation (2.24). The reconstruction loss ($\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$) enforces a high similarity of input data and generated data. At the same time, the Kullback-Leibler divergence ($D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}))$) regularizes the latent space such that the encoded input data follow the prior distribution. Consequently, the VAE is effectively only trained on latent points $\mathbf{z} \in \mathcal{Z}$ with a high probability $p_\theta(\mathbf{z})$. Therefore, decoding latent points carrying a low probability can result in decoded data outside the training data distribution, leading to low-quality or invalid samples (e.g., invalid architectures in a predefined search space). The region of latent points $\mathbf{z}$, which can be decoded to data following the training data distribution is denoted in Tripp et al. (2020) as the *feasible region* $\mathcal{Z}' \subset \mathcal{Z}$.

Thus, the first shortcoming in the mentioned two-step approach is that most training data are low-scoring data (e.g., networks with a low accuracy on a downstream task) and thereby $\mathcal{Z}'$ is mainly filled by low-scoring points. This hinders the optimization, since many novel, high-scoring points are outside this feasible region. Another shortcoming is that learning a latent space, which captures the training data distribution well, has a different objective than learning a latent space in which we can apply efficient optimization to find high-scoring data. Lastly, information about new points found during the optimization is not propagated back to the generative model to adjust the latent space and its feasible region.

5.1.2 *Weighted Retraining*

In order to solve the mentioned drawbacks of the two-step optimization in the latent space, Tripp et al. (2020) propose to *directly optimize the latent space* with weighted retraining and show its ability exemplary using a VAE w.r.t. a target function to adapt the latent space for the optimization of images and arithmetic functions using BO.

The intuition of this approach is to place more probability mass on high-scoring latent points (e.g., high-performing or low latency architectures) and less probability mass on low-scoring points. The scoring is given by the objective function $f\colon \mathcal{X} \to \mathbb{R}$. In the settings analyzed so far in this thesis, this can be the validation performance of architectures $\mathbf{x} \in \mathcal{X}$ as in Section 2.3.1. This strategy does not discard low-scoring architectures completely, which would be inadequate for proper learning. The generative model is therefore trained on a data distribution that systematically increases the probability of high-scoring latent points. This can be done by simply assigning a weight w_n to each data point $\mathbf{x}_n \in D$, $D = \{(\mathbf{x}_n, f(\mathbf{x}_n))\}_{n=1}^N$. This weight indicates the likelihood of the data point to occur during batch-wise training. Accordingly, the total loss is obtained as the weighted mean of the local losses for each data point. Note, in the case of a VAE, the loss corresponds to the ELBO as presented in Section 2.3.2, Equation (2.24). Concerning the weight

specification, Tripp et al. (2020) propose a rank-based weight function

$$w(\mathbf{x}; D, k) \propto \frac{1}{kN + \text{rank}_{f,D}(\mathbf{x})}$$
$$\text{rank}_{f,D}(\mathbf{x}) = |\{\mathbf{x}_n \in D : f(\mathbf{x}_n) > f(\mathbf{x})\}|. \tag{5.1}$$

The weight is always positive with a tunable hyperparameter k.

The proposed weighted training allows now for a coupling of generation and optimization objectives. In order to enable for updating the latent space and the feasible region, Tripp et al. (2020) propose periodic retraining, i.e., retraining the generative model (e.g, VAE) during the optimization itself.

5.2 ARCHITECTURE GENERATIVE MODEL

Our proposed model, AG-Nets's generator, is inspired by the model SVGe, presented in Chapter 4, with the aim to inherit its flexible applicability to various search spaces. Yet, similar to Yan et al. (2020), due to the intrinsic discretization and training setting, SVGe does not allow for backpropagation. In this chapter, we propose a GNN generator, which circumvents the intermediate architecture discretization and can therefore be trained by a single reconstruction loss using backpropagation. Its iterative optimization is inspired by Tripp et al. (2020), as presented in Section 5.1.2. Our model transfers the idea of weighted retraining to NAS. It uses our plain generator and improves sample efficiency by employing a differentiable surrogate model on the target function such that, in contrast to Tripp et al. (2020), no further black-box optimization step is needed. Next, we describe the proposed generator network and surrogate model.

5.2.1 *Preliminaries*

Recall, we aim to generate neural networks represented as DAGs, $G = (V, E)$, with nodes $v \in V$ and edges $e \in E$. In contrast to chapter 4 the nodes v are represented as one-hot vectors of the node types. The graph representations differ between the various benchmarks in terms of their labeling of operations. For example in NAS-Bench-101 (Ying et al., 2019) each node is associated with an operation, whereas in NAS-Bench-201 (Dong and Yang, 2020) each edge is associated with an operation.

5.2.2 *Generative Network*

Commonly used graph generative networks are based on VAE. In contrast, our proposed network is a *purely generative* network, p_G (see Figure 5.2). To generate valid graphs, we build our model similar to the graph decoder from the VAE approach SVGe, Section 4.2.2. The generator takes a randomly sampled variable $\mathbf{z} \sim \mathcal{N}(0, 1)$ as input and reconstructs a randomly sampled graph from the cell-based search space. The model iteratively builds the graph: it starts with generating the input node v_0, followed by adding subsequent nodes v_i and their labels and connecting them with edges $e_{(j,i)}$, $j < i$, until the end node v_T with the label *output* is generated. Additionally, we want to learn a surrogate for performance prediction on the generated data and allow for end-to-end training of both. To allow for backpropagation, we need to adapt several details of the generator model. We initialize the node-attributes for each node by one-hot encoded vectors, which are

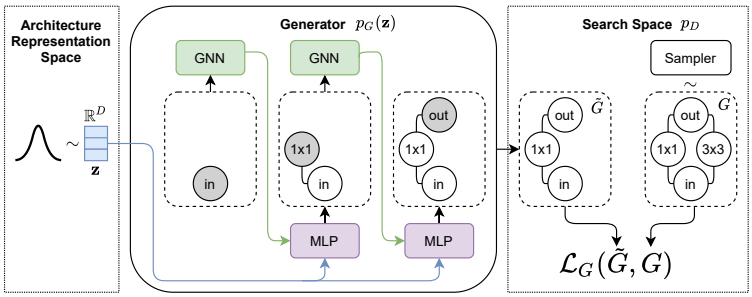


Figure 5.2: Representation of the training procedure for our generator in AG-Net. The input is a randomly sampled latent vector $\mathbf{z} \in \mathbb{R}^d$. First, the input node is generated, initialized and input to a GNN to generate a partial graph representation. The learning process iteratively generates node scores and edge scores using $\mathbf{z}$ and the partial graph representation until the output node is generated. The target for this generated graph is a randomly sampled architecture.

initialized during training using a 2-layer MLP to replace the learnable look-up table proposed in SVGe (Section 4.2.2). The output of our generator is a vector graph representation consisting of a concatenation of generated node scores and edge scores. The node score for a node $v \in V$ is a distribution over all possible node types. Sampling from this (categorical) distribution yields a one-hot encoding of a specific node type. The edge score is a distribution over the existence of the edge, which is passed into a Bernoulli distribution at inference time. The sampled node scores for all nodes in a graph represent the node attribute matrix, while the sampled edge scores represent the upper triangular adjacency matrix.

It is important to note that the iterative generation process is independent of the ground truth data, which are only used as a target for the reconstruction loss. Note that the end-to-end trainability of the proposed generator is a prerequisite for our model: It allows to pair the generator with a learnable performance predictor such that information on the expected architectures' scoring target as the accuracy can be learned by the generator. This enables a stronger coupling with the predictor's target for the generation process and higher query efficiency (see Section 5.4). In contrast, previous models such as Huang and Chu (2021) and Yan et al. (2020) are not fully differentiable and do not allow such optimization. Our generative model is pretrained on the task of reconstructing neural architectures, where for each randomly drawn latent space sample, we evaluate the reconstruction loss to a randomly drawn architecture. This simple procedure is facilitated by the heavily constrained search spaces of neural architectures, making it easy for the model to learn to generate valid architectures without being supported by a discriminator model as in generative adversarial networks (GANs) (Goodfellow et al., 2014).

The pseudo algorithm for the graph generation is described in Algorithm 5. The modules f_{initNode}, f_{addNode}, f_{addEdges}, $f_{\text{Embedding}}$ used in this code are two-layer MLPs with ReLU activation functions. Note, in contrast to SVGe, we don't sample within the generation process, in order to allow for end-to-end learning with the prediction model for AG-Net.

5.2.3 *Performance Predictor*

The presented generative model is coupled with a simple surrogate model, a 4-layer MLP with ReLU non-linearities, for target predictions C. The hidden size equals the input size. These targets can be

Algorithm 5: Graph Generation

Input: $\mathbf{z} \sim \mathcal{N}(0,1)$
Output: reconstructed graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$

initialize one-hot encoded InputNode v_0, with embedding $\mathbf{h}_0 \leftarrow f_{\text{initNode}}(\mathbf{z}, f_{\text{Embedding}})[\text{InputType}])$
$s_V \leftarrow \{v_0\}, s_E \leftarrow \emptyset, E \leftarrow \emptyset, \mathbf{h}_G \leftarrow \mathbf{z},$
while $|V| \leq$ *Max Number of Nodes* **do**
  $v_{t+1} \leftarrow f_{\text{addNode}}(\mathbf{z}, \mathbf{h}_G)$
  $s_V \leftarrow s_V \cup \{v_{t+1}\}$
  $\mathbf{h}_{t+1} \leftarrow f_{\text{initNode}}(\mathbf{z}, \mathbf{h}_G, f_{\text{Embedding}}(v_{t+1})])$
  for $v_j \in V \setminus v_{t+1}$ **do**
    $s_{\text{addEdges}}(j, t+1) \leftarrow f_{\text{addEdges}}(\mathbf{h}_{t+1}, \mathbf{h}_t, \mathbf{h}_G, \mathbf{z})$
    $e_{(j,t+1)} \sim \text{Eval}(s_{\text{addEdges}}(j, t+1))$; ▷ evaluate whether to add edge
    if $e_{(j,t+1)} = 1$ **then**
      $s_E \leftarrow s_E \cup \{s_{\text{addEdges}}(j, t+1)\}$
      $E \leftarrow E \cup \{e_{(j,t+1)} = (v_j, v_{t+1})\}$
    end
  end
  $\mathbf{h}_t \leftarrow \text{concat}(\mathbf{h}_t, \mathbf{h}_{t+1})$
  $G \leftarrow (s_V, E)$
  $\mathbf{h}_t \leftarrow (\mathbf{h}_t, G)$; ▷ update node embeddings Eq. (4.1), (4.2)
  $\mathbf{h}_G \leftarrow \text{aggregate}(\mathbf{h}_t)$; ▷ update graph embedding Eq. (4.3), (4.4)
  $t \leftarrow t + 1$
end
$\widetilde{V} \sim \text{Categorical}(s_V)$; ▷ Sample node types
$\widetilde{E} \sim \text{Ber}(s_E)$; ▷ Sample edges
$\widetilde{G} = (\widetilde{V}, \widetilde{E})$

validation or test accuracy of the generated graph, or the latency with respect to a certain hardware. Our AG-Net passes the output of our generator, i.e., a generated vector representation, as the direct input to our MLP surrogate model. The generated vector representation is a concatenation of the flatted generated node score matrix and flatted upper triangular matrix of the adjacency matrix, presented by the generated edge scores. Note the scores of the generated graph are given by s_V, s_E in Algorithm 5 Line 21 and Line 22 before the sampling.

For comparison, we also include a tree-based method, XGBoost (XGB) (Chen and Guestrin, 2016) as an alternative prediction model. XGB is used as a performance surrogate model in NAS-Bench-301 (Zela et al., 2022) and shows high prediction abilities. The input to XGB is the binary vector representation of the architectures, i.e., the one-hot encoded flatted node attribute matrix and the binary upper triangular adjacency matrix. Since this method is non-differentiable, we additionally include a gradient estimation for rank-based metrics (Rolínek et al., 2020). This way, we are able to include gradient information to the generator. Yet, it is important to note, that this approach is not fully differentiable. This comparison will allow us to measure the trade-off between using supposedly stronger predictors over the capability to allow for full end-to-end learning.

For visualizations of the representations used in this chapter for the different search spaces, see Appendix A. Note, the vector representation dimension differs across the search spaces due to the different maximal amount of nodes.

5.2.4 *Training Objectives*

The generative model p_G learns to reconstruct a randomly sampled architecture G from search space p_D given a randomly sampled latent vector $\mathbf{z} \sim \mathcal{N}(0,1)$. The objective function for this generation process can be formulated as the sum of node-level loss $\mathcal{L}_V$ and edge-level loss $\mathcal{L}_E$:

$$\mathcal{L}_G(\widetilde{G}, G) = \mathcal{L}_V + \mathcal{L}_E;\ \widetilde{G} \sim p_G(\mathbf{z});\ G \sim p_D, \tag{5.2}$$

where $\mathcal{L}_V$ is the Cross-Entropy loss between the predicted and the ground truth nodes and $\mathcal{L}_E$ is the Binary Cross-Entropy loss, between the predicted and ground truth edges of the generated graph $\widetilde{G}$. This training step is *completely unsupervised*. Figure 5.2 presents an overview of the training process.

To include the training of the surrogate model, the objective function is reformulated to:

$$\mathcal{L}(\widetilde{G}, G) = (1-\alpha)\mathcal{L}_G(\widetilde{G}, G) + \alpha\mathcal{L}_C(\widetilde{G}, G), \tag{5.3}$$

where α is a hyperparameter to trade-off generator loss $\mathcal{L}_G$ and prediction loss $\mathcal{L}_C$ for the prediction targets C of graph G. We set the predictor loss as an MSE. Furthermore, each loss is optimized using mini-batch gradient descent.

5.2.5 *Generative Latent Space Optimization*

To facilitate the generation process, we optimize the architecture representation space via weighted retraining as presented in Section 5.1.2, resulting in a sample efficient search algorithm. In the following we describe the weighted retraining again using our generative model.

Given data $G_i \sim p_D$, we can write Equation (5.1) as:

$$\begin{aligned} w(G; p_D, k) &\propto \frac{1}{kN + \text{rank}_{f,p_D}(G)} \\ \text{rank}_{f,p_D}(G) &= |\{G_i : f(G_i) > f(G), G_i \sim p_D\}|, \end{aligned} \tag{5.4}$$

where $f(\cdot)$ is the evaluation function of the architecture G_i; for NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201 (Dong and Yang, 2020) it is the tabular benchmark entry, for NAS-Bench-301 (Zela et al., 2022) and NAS-Bench-NLP (Klyuchnikov et al., 2022) it is the surrogate benchmark prediction. Similar to Tripp et al. (2020), we set $k = 10e-3$. The retraining procedure itself then consists of fine-tuning the pretrained generative model coupled with the surrogate model, where loss functions and data points are both weighted by $w(G; p_D, k)$.

A more descriptive visualization of this latent space optimization technique is displayed in Figure 5.3.

5.2.6 *AG-Net Search Process*

So far, we have presented all the individual components for the search algorithm which we propose in this chapter. Next, we will describe it in more detail.

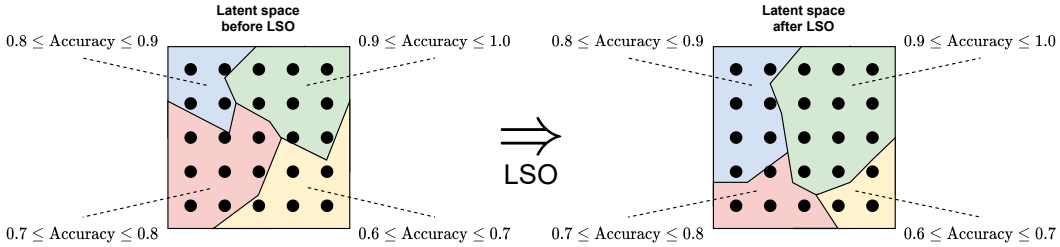


Figure 5.3: The latent space is reshaped in a way that promotes desired properties of generated architectures (in this example: accuracy). Consequently, it becomes more likely for the generator to generate architectures satisfying this property.

We first need to pretrain our generative model on the architecture search space as presented in Section 5.2.2 and Section 5.2.4. Our NAS algorithm is then initialized by randomly sampling 16 architectures from the architecture search space, which are then weighted by the weighting function $\mathcal{W} = w(G)_{G \sim p_D}$. Then, latent space optimized architecture search is performed by iteratively retraining the generator coupled with the surrogate model for e epochs and generating 100 architectures of which the top-16 (according to their target prediction) are evaluated and added to the training data. This step is repeated until the desired number of queries is reached. When generating architectures, we sample from a grid $\mathbf{z} \sim \mathfrak{U}[-3, 3]$, containing the 99%-quantiles from $\mathcal{N}(0, 1)$ uniformly distributed. This way, we sample more distributed latent variables for better latent space coverage. We provide a high-level description of the search in Algorithm 6.

5.3 EXPERIMENTS

We evaluate the proposed simple architecture generative network (AG-Net) on the two commonly used tabular benchmarks NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201 (Dong and Yang, 2020), the surrogate benchmarks NAS-Bench-301 (Zela et al., 2022) evaluated on the DARTS search space (Liu et al., 2019), NAS-Bench-NLP (Klyuchnikov et al., 2022) and the first hardware device induced benchmark Hardware-Aware Benchmark (Li et al., 2021a). Additionally, we perform experiments on the ImageNet (Deng et al., 2009) classification task and show state-of-the-art performance on the DARTS search space. In our experiments in Section 5.3.4 for the Hardware-Aware Benchmark we consider the latency information on the NAS-Bench-201 search space. Details about all hyperparameters are given Appendix B.3.

5.3.1 *Experiments on Tabular Benchmarks*

NAS-Bench-101

For our experiments on NAS-Bench-101, we first pretrain our generator for generating valid graphs on the NAS-Bench-101 search space. This step does not require information about the performance of architectures and is therefore inexpensive. The pretrained generator is then used for all experiments on NAS-Bench-101. We retrain this generator in a weighted manner jointly with the surrogate model for 15 epochs. The search proceeds as described in Section 5.2.6. We compare our method to the VAE-based search method arch2vec (Yan et al., 2020) and predictor-based model

Algorithm 6: Unconstrained Search Algorithm

Input: (i) Search space p_D
Input: (ii) Pretrained generator G
Input: (iii) Untrained performance predictor P
Input: (iv) Query budget b
Input: (v) e epochs to train G and P
▷ `Initialize training data`
$\mathbf{D} \leftarrow \{\}$
while $|\mathbf{D}| < 16$ **do**
  $\mathbf{D} \leftarrow \mathbf{D} \cup \{d \sim p_D\}$
end
▷ `Evaluate architectures (get accuracies on target image dataset)`
$\mathbf{D} \leftarrow \text{eval}(\mathbf{D})$
▷ `Randomly initialize predictor weights`
$P \leftarrow \text{init}(P)$
▷ `Search loop`
while $|\mathbf{D}| < b$ **do**
  ▷ `Weight training data by performance`
  $\mathbf{D}_\text{w} \leftarrow \text{weight}(\mathbf{D})$
  ▷ `Train generator and predictor`
  $\text{train}(G, P, \mathbf{D}_\text{w}, e)$
  ▷ `Generate 100 candidates`
  $\mathbf{D}_\text{cand} \leftarrow \{\}$
  while $|\mathbf{D}_{cand}| < 100$ **do**
    $\mathbf{z} \sim \mathfrak{U}[-3, 3]$
    $\mathbf{D}_\text{cand} \leftarrow \mathbf{D}_\text{cand} \cup G(\mathbf{z})$
  end
  ▷ `Select top 16 candidates with P`
  $\mathbf{D}_\text{cand} \leftarrow \text{select}(\mathbf{D}_\text{cand}, P, 16)$
  ▷ `Evaluate and add to data`
  $\mathbf{D} \leftarrow \mathbf{D} \cup \text{eval}(\mathbf{D}_\text{cand})$
end

WeakNAS (Wu et al., 2021a), as well as state-of-the-art methods, such as NAO (Luo et al., 2018)[3], random search (Li and Talwalkar, 2019), local search (White et al., 2021b), Bayesian optimization with DNGO (Snoek et al., 2015), regularized evolution (Real et al., 2019) and BANANAS (White et al., 2021a)[4]. Additionally, we compare the proposed AG-Net to the model using an XGB predictor, as introduced in Section 5.2.3. The results of this comparison are listed in Table 5.1. Here, we report the mean over 10 runs. Note, we search for the architecture with the best validation accuracy and report the corresponding test accuracy. Furthermore, we plot the search progress in Figure 5.4 (top left). As we can see, our model AG-Net improves over all state-of-the-art methods, not only at

[3] We reran this experiment using the implementation from White et al. (2021c).

[4] We reran these experiments using the official implementation from White et al. (2020), White et al. (2021a), and White et al. (2021b), with the same initial training data and amount of top k architectures as for AG-Net.

NAS Method	Val. Acc. (%) ↑	Test Acc. (%) ↑	Queries
Optimum*	95.06	94.32	
arch2vec + RL (Yan et al., 2020)	-	94.10	400
arch2vec + BO (Yan et al., 2020)	-	94.05	400
NAO [3] (Luo et al., 2018)	94.66	93.49	192
BANANAS[4] (White et al., 2021a)	94.73	94.09	192
Bayesian Optimization[4] (Snoek et al., 2015)	94.57	93.96	192
Local Search[4] (White et al., 2021b)	94.57	93.97	192
Random Search[4] (Li and Talwalkar, 2019)	94.31	93.61	192
Regularized Evolution[4] (Real et al., 2019)	94.47	93.89	192
WeakNAS (Wu et al., 2021a)	-	**94.18**	200
XGB (ours)	94.61	94.13	192
XGB + ranking (ours)	94.60	94.14	192
AG-Net (ours)	**94.90**	**94.18**	192

Table 5.1: Results on NAS-Bench-101 for the search of the best architecture in terms of validation accuracy on CIFAR-10 to state-of-the-art methods (mean over 10 trials).

the last query of 300 data points, reaching a top-1 test accuracy of 94.2%, but is also almost any time better during the search process.

A direct comparison to the recently proposed GANAS (Rezaei et al., 2021) on NAS-Bench-101 is difficult, since GANAS searches on NAS-Bench-101 until they find the best architecture in terms of validation accuracy, whereas we limit our search to a maximal amount of 192 queries and are able to find high-performing architectures already in this small query setting. The comparison of AG-Net to the generator paired with an XGBoost (Chen and Guestrin, 2016) predictor shows that our end-to-end learnable approach is favorable even over potentially stronger predictors.

NAS-Bench-201

This benchmark contains three different image classification tasks: CIFAR-10, CIFAR-100 (Krizhevsky, 2009) and ImageNet16-120 (Chrabaszcz et al., 2017). For the experiments on NAS-Bench-201 we retrain AG-Net in the weighted manner for 30 epochs (see Section 5.2.6). In this setting, we also compare AG-Net to two recent generative models (Rezaei et al., 2021; Huang and Chu, 2021). SGNAS (Huang and Chu, 2021) trains a supernet by uniform sampling, following Dong and Yang (2019a). Additionally, a CNN-based architecture generator is trained to search architectures on the supernet. When comparing with Yan et al. (2020), we also adopt their evaluation scheme of adding only the best-performing architecture found so far (top-1) to the training data instead of top-16 as in our other experiments.

We report the search results for different numbers of queries for the NAS-Bench-201 dataset in Table 5.2. In addition, we plot the search progress in terms of queries in Figure 5.4 (top right, middle). Our method provides state-of-the-art results on all datasets for a varying number of queries. Most importantly, AG-Net shows strong performance in the few-query regime compared to Yan et al. (2020) except for CIFAR-100, proving its high query efficiency.

NAS Method	CIFAR-10		CIFAR-100		ImageNet16-120		Queries	Search Method
	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑		
Optimum*	91.61	94.37	73.49	73.51	46.77	47.31		
SGNAS (Huang and Chu, 2021)	90.18	93.53	70.28	70.31	44.65	44.98		Supernet
arch2vec + BO (Yan et al., 2020)	91.41	94.18	**73.35**	**73.37**	46.34	46.27	100	Bayesian Optimization
AG-Net (ours)	**91.55**	**94.24**	73.2	73.12	46.31	46.2	96	Generative LSO
AG-Net (ours, topk=1)	91.41	94.16	73.14	73.15	**46.42**	**46.43**	100	Generative LSO
BANANAS[4] (White et al., 2021a)	91.56	94.3	**73.49***	73.50	**46.65**	**46.51**	192	Bayesian Optimization
BO[4] (Snoek et al., 2015)	91.54	94.22	73.26	73.22	46.43	46.40	192	Bayesian Optimization
RS [4] (Li and Talwalkar, 2019)	91.12	93.89	72.08	72.07	45.87	45.98	192	Random
XGB (ours)	91.54	94.34	73.10	72.93	46.48	46.08	192	Generative LSO
XGB + Ranking (ours)	91.48	94.25	73.20	73.24	46.40	46.16	192	Generative LSO
AG-Net (ours)	**91.60**	**94.37***	**73.49***	**73.51***	46.64	46.43	192	Generative LSO
GANAS (Rezaei et al., 2021)	-	94.34	-	73.28	-	**46.80**	444	Generative Reinforcement Learning
AG-Net (ours)	**91.61***	**94.37***	**73.49***	**73.51***	**46.73**	46.42	400	Generative LSO

Table 5.2: Architecture Search on NAS-Bench-201. We report the mean over 10 trials for the search of the architecture with the highest validation accuracy.

5.3.2 *Experiments on Surrogate Benchmarks*

We furthermore apply our search method on larger search spaces as DARTS and NAS-Bench-NLP without ground truth evaluations for the whole search space, making use of surrogate benchmarks as NAS-Bench-301 (Zela et al., 2022), NAS-Bench-x11 (Yan et al., 2021) and NAS-Bench-Suite (Mehta et al., 2022).

NAS-Bench-301

Here, we report experiments on the cell-based DARTS search space using the surrogate benchmark NAS-Bench-301 for the CIFAR-10 image classification task. The results are described in Table 5.3 (left) and visualized in Figure 5.4 (bottom left). Our method is comparable to other state-of-the-art methods in this search space. Since the DARTS search space is defined by two different cells, normal and reduction cell, we have to modify the search process described in Section 5.2.6. In the following, we will describe the exact search procedure using the different cells individually.

Search Process using NAS-Bench-301 For experiments in the DARTS (Liu et al., 2019) search space, we first train our generative model on generating valid cells (see Figure A.3 for a visualization). For this we do not distinguish between generating a normal or a reduction cell. Eventually, having a pretrained generative model for generating valid cell representations in the DARTS search space allows for searching well-performing architectures. The next step is the search for architectures by means of the surrogate benchmark NAS-Bench-301 (Zela et al., 2022), which trained and evaluated 60 000 architectures on CIFAR-10 and provides predicted performances of architectures in the DARTS search space. We begin the search by randomly sampling 16 architectures from NAS-Bench-301. Next, we generate one random normal cell. This cell is used to search for the best reduction cell, by iteratively generating reduction cells. For that we combine the generated normal cell and the generated reduction cells for accuracy evaluations using the surrogate benchmark NAS-Bench-301. This search procedure then follows the same steps as described in Section 5.2.6 and stops after we reach a query amount of 155. Now, we can use the best found reduction cell as a fixed starting point to search for the best normal cell in the

NAS Method	NAS-Bench-301		NAS-Bench-NLP	
	Val. Acc.(%) ↑	Queries	Val. Perplexity (%) ↑	Queries
BANANAS[4] (White et al., 2021a)	94.77	192	95.68	304
Bayesian Optimization[4] (Snoek et al., 2015)	94.71	192	-	-
Local Search[4] (White et al., 2021b)	**95.02**	192	95.69	304
Random Search[4] (Li and Talwalkar, 2019)	94.31	192	95.64	304
Regularized Evolution[4] (Real et al., 2019)	94.75	192	95.66	304
XGB (ours)	94.79	192	**95.95**	304
XGB + Ranking (ours)	94.76	192	95.92	304
AG-Net (ours)	94.79	192	95.86	304

Table 5.3: Results on: **(left)** NAS-Bench-301 (mean validation accuracy over 50 trials). **(right)** NAS-Bench-NLP (mean validation perplexity over 100 trials).

same manner as before. The overall search stops after a maximal amount of 310 queries. The search outcome differs between starting with a reduction or the normal cell. Note, the search procedure starting with a random reduction cell is analogous. In this chapter, we start with a random reduction cell.

NAS-Bench-NLP

Next, we evaluate AG-Net on NAS-Bench-NLP (Klyuchnikov et al., 2022) for the language modeling task on Penn TreeBank (Mikolov et al., 2010). We retrain AG-Net coupled with the surrogate model for 30 epochs to predict the validation perplexity. Note, since the search space considered in NAS-Bench-NLP is too large for a full tabular benchmark evaluation, we make use of the surrogate benchmark NAS-Bench-x11 (Yan et al., 2021) and NAS-Bench-Suite (Mehta et al., 2022) instead of tabular entries, which provides surrogate predictions for NAS-Bench-NLP. More information are presented in Appendix A.4.

We compare our methods to the same state-of-the-art methods as in previous experiments. The results are reported in Table 5.3 (right) and visualized in Figure 5.4 (bottom right). Our AG-Net improves over all state-of-the-art methods by a substantial margin and using XGB as a predictor even improves the search further.

5.3.3 *ImageNet Experiments*

The previous experiment on NAS-Bench-301 shows the ability of our generator to generate valid architectures and to perform well in the DARTS search space. This allows for searching for a well-performing architecture on ImageNet (Deng et al., 2009). Yet evaluating up to 300 different found architectures on ImageNet is extremely expensive. Therefore, we consider two approaches using predictor-based methods and training-free methods. Our first approach is to retrain the best found architectures on the CIFAR-10 from the previous experiment on NAS-Bench-301 (AG-Net and the XGB adaptions) on ImageNet. Our second approach is based on a training-free neural architecture search approach. The recently proposed TE-NAS (Chen et al., 2021) provides a training-free neural architecture search approach, by ranking architectures by analyzing the neural tangent kernel by

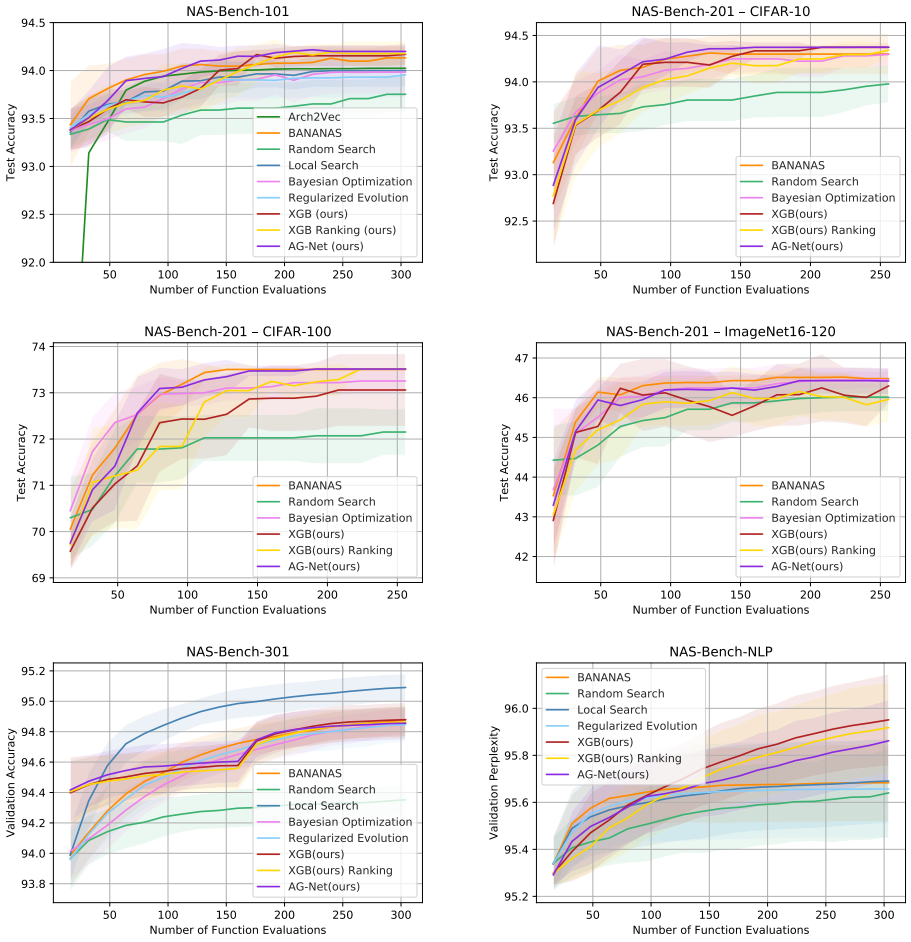


Figure 5.4: Architecture search evaluations on NAS-Bench-101, NAS-Bench-201, NAS-Bench-301 and NAS-Bench-NLP for different search methods. Accuracy and perplexity in %.

its condition number (CN) and the number of linear regions (NLR) of each architecture. These two measurements are training-free and do not need any labels. The intuition between those two measurements is their implication towards trainability and expressivity of a neural architecture and also their correlation with the neural architecture's accuracy; CN is negatively correlated and NLR positively correlated with the architecture's test accuracy. We adapt this idea for our search on ImageNet and search architectures in terms of their CN and NLR instead of their validation accuracy.

Table 5.4 shows the results. Note that our latter described search method on ImageNet is **training-free** (as TE-NAS (Chen et al., 2021)) and the amount of queries displays the amount of data we evaluated for the training-free measurements. Other query information include the amount of (partly) trained architectures. Furthermore, the displayed differentiable methods are based on training supernets which can lead to expensive training times. For the predictor-based search approach, the best found architectures on NAS-Bench-301 (CIFAR-10) result in comparable error rates on ImageNet to former approaches. As a result, our search method approach is highly efficient and outperforms previous methods in terms of needed GPU days. The result in terms

NAS Method	Top-1↓	Top-5↓	# Queries	Search GPU days
Mixed Methods				
NASNET-A (CIFAR-10) (Zoph et al., 2018)	26.0	8.4	20 000	2 000
PNAS (CIFAR-10) (Liu et al., 2018a)	25.8	8.1	1 160	225
NAO (CIFAR-10) (Luo et al., 2018)	24.5	7.8	1 000	200
Differentiable Methods				
DARTS (CIFAR-10) (Liu et al., 2019)	26.7	8.7	-	4.0
SNAS (CIFAR-10)(Xie et al., 2019b)	27.3	9.2	-	1.5
PDARTS (CIFAR-10) (Chen et al., 2019)	**24.4**	**7.4**	-	0.3
PC-DARTS (CIFAR-10) (Xu et al., 2020)	25.1	7.8	-	**0.1**
PC-DARTS (ImageNet) (Xu et al., 2020)	**24.2**	**7.3**	-	3.8
Predictor-based Methods				
WeakNAS (ImageNet) (Wu et al., 2021a)	**23.5**	**6.8**	800	2.5
XGB (NB-301)(CIFAR-10) (ours)	24.1	7.4	304	**0.02**
XGB + Ranking (NB-301)(CIFAR-10) (ours)	24.1	7.2	304	**0.02**
AG-Net (NB-301)(CIFAR-10) (ours)	24.3	7.3	304	0.21
Training-Free Methods				
TE-NAS (CIFAR-10) (Chen et al., 2021)	26.2	8.3	-	0.05
TE-NAS (ImageNet) (Chen et al., 2021)	24.5	7.5	-	0.17
AG-Net (CIFAR-10) (ours)	**23.5**	7.1	208	**0.02**
AG-Net (ImageNet) (ours)	**23.5**	6.9	208	**0.09**

Table 5.4: ImageNet **error** of neural architecture search on DARTS.

of top-1 and top-5 error rates are even improving over the one from previous approaches when using the training-free approach.

As we described in the previous section, the search in the DARTS search space using NAS-Bench-301 needs adaptions in the search procedure. We describe the further adaption of using training-free measurements instead of the NAS-Bench-301 prediction in the following.

Search Process using TE-NAS Concretely, for the search on ImageNet we search for architectures in terms of their CN and NLR instead of their validation accuracy. In the beginning of our search we generate three random normal cells. These cells are used to search for an optimal reduction cell optimizing both CN and NLR measurements. In each search iteration we generate reduction cells and calculate the CN and NLR for each combination of normal cell and reduction cell. The reduction cells are ranked according to their mean CN and their mean NLR (mean in terms of all three normal cells). The 16 best ranked reduction cells are then used for the next iteration of reduction cell search. The reduction cell search stops, when a maximum of 104 queries is reached. After that we use the best found reduction cell in terms of the lowest CN and the highest NLR for the next search for a normal cell. The next steps use this best found reduction cell as a starting point and searches for the best normal cell in the same manner as before. The search

Settings		**Best out of 10 runs**						**Mean**							
Constraint		**Joint=0**		**Joint=1**		**Random**		**Joint=0**		**Joint=1**		**Random**		**Optimum***	
Device	Lat.↓	Acc.↑	Lat.↓	Acc.↑	Lat.↓	Acc.↑	Lat.↓	Acc.↑	Feas.↑	Acc.↑	Feas.↑	Acc.↑	Feas.↑	Acc.↑	Lat.↓
Edge GPU	2	**0.406***	1.90	**0.406***	1.90	0.397	1.78	**0.397**	0.29	*0.391*	0.31	0.372	0.05	0.406	1.90
Edge GPU	4	**0.448***	3.49	**0.448***	3.49	0.437	3.35	*0.428*	0.29	**0.433**	0.43	0.417	0.22	0.448	3.49
Edge GPU	6	*0.458*	5.29	**0.464***	5.96	*0.458*	5.29	**0.453**	0.64	*0.450*	0.79	0.449	0.72	0.464	5.96
Edge GPU	8	*0.465*	6.81	**0.468***	6.81	0.464	7.44	**0.463**	0.98	*0.462*	0.99	0.457	1.00	0.468	6.81
Raspi 4	2	**0.355***	1.58	**0.355***	1.58	0.348	1.60	*0.346*	0.28	**0.347**	0.30	0.339	0.08	0.355	1.58
Raspi 4	4	*0.431*	3.83	**0.436***	3.79	0.427	3.85	*0.420*	0.47	**0.428**	0.50	0.419	0.37	0.436	3.79
Raspi 4	6	*0.449*	5.95	**0.452***	5.29	0.445	5.95	*0.440*	0.56	**0.441**	0.57	0.432	0.55	0.452	5.29
Raspi 4	8	*0.456*	6.33	0.455	7.96	**0.457**	7.97	**0.451**	0.69	*0.449*	0.79	0.447	0.76	0.465	7.43
Raspi 4	10	**0.466**	8.66	*0.465*	8.62	0.464	8.72	**0.464**	0.77	*0.454*	0.94	*0.454*	0.90	0.468	8.83
Raspi 4	12	**0.468***	8.83	0.463	9.05	*0.464*	8.72	**0.465**	0.91	*0.457*	0.98	0.456	0.96	0.468	8.83
Edge TPU	1	**0.468***	0.96	*0.466*	0.97	0.464	1.00	**0.464**	0.74	*0.457*	0.82	0.454	0.79	0.468	0.96
Pixel 3	2	**0.413***	1.30	**0.413***	1.30	0.400	1.50	**0.409**	0.48	*0.405*	0.59	0.388	0.30	0.413	1.30
Pixel 3	4	**0.460***	3.55	0.446	3.01	*0.447*	3.23	**0.453**	0.69	*0.441*	0.77	0.438	0.64	0.460	3.55
Pixel 3	6	*0.464*	5.92	**0.465***	5.95	0.458	4.68	**0.457**	0.77	*0.452*	0.94	0.451	0.88	0.465	5.57
Pixel 3	8	**0.468***	6.65	*0.465*	7.88	0.461	7.13	**0.464**	0.87	*0.457*	0.99	0.454	0.97	0.468	6.65
Pixel 3	10	**0.466**	6.70	0.461	8.48	*0.464*	8.01	**0.464**	0.96	0.455	1.00	*0.456*	0.99	0.468	6.65
Eyeriss	1	**0.452***	0.98	*0.449*	0.98	0.447	0.98	**0.445**	0.49	*0.436*	0.53	0.433	0.23	0.452	0.98
Eyeriss	2	**0.465**	1.65	**0.465**	1.65	0.464	1.65	**0.463**	0.87	*0.457*	0.99	*0.457*	0.95	0.468	1.65
FPGA	1	**0.440**	1.00	**0.440**	0.97	0.438	0.97	**0.433**	0.65	**0.433**	0.80	0.429	0.58	0.444	1.00
FPGA	2	**0.465***	1.60	0.460	1.60	*0.463*	1.97	**0.462**	0.82	0.451	0.99	*0.453*	0.97	0.465	1.60

Table 5.5: Results for searches with at most 200 queries on HW-NAS-Bench (Li et al., 2021a) with varying devices and latency (Lat.) constraints in two multi-objective settings: *Joint=0* optimizes accuracy under latency constraint, while *Joint=1* optimizes for accuracy and latency jointly. We report the best found architecture out of 10 runs with their corresponding latency, as well as the mean of these runs. We compare to random search as a strong baseline (Li and Talwalkar, 2019). Feasibility (Feas.) is the proportion of evaluated architectures during the search that satisfy the latency constraint (larger is better). The optimal architecture (*****) is the architecture with the highest accuracy satisfying the latency constraint.

stops after a total of 208 queries and outputs an overall normal and reduction cell combination, leading to a DARTS architecture, which we train on ImageNet using the same training pipeline as Chen et al. (2021). Note, we use here three generated random cells as starting point, since only one single minibatch is used to calculate the CN, which can lead to noisy behavior.

5.3.4 *Experiments on Hardware-Aware Benchmark*

Next, we apply AG-Net to the Hardware-Aware NAS-Benchmark (Li et al., 2021a). We demonstrate in two settings that AG-Net can be used for multi-objective learning. The first setting (*Joint=1*) is formulated as constrained joint optimization, for any $h \in H$:

$$\begin{aligned} &\max_{G \sim p_D} f(G) \wedge \min_{G \sim p_D} g_h(G) \\ &\text{s.t. } g_h(G) \leq L, \end{aligned} \tag{5.5}$$

where $f(\cdot)$ evaluates architecture G for accuracy and $g_h(\cdot)$ evaluates for latency given a hardware $h \in H$ and a user-defined latency constraint L. The second setting (*Joint=0*) is formulated as

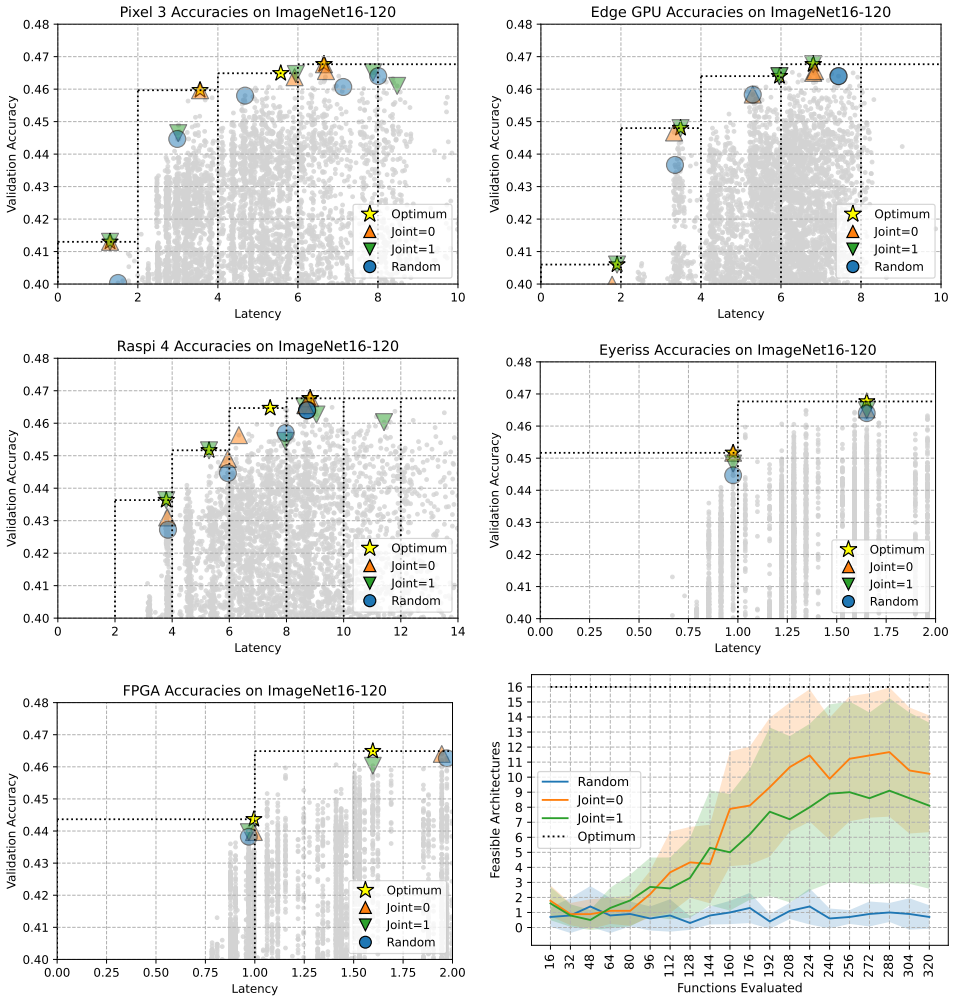


Figure 5.5: (**top left to bottom left)** Exemplary searches on HW-NAS-Bench for image classification on ImageNet16-120 with 192 queries on Pixel 3, Edge GPU, Raspi 4, Eyeriss, FPGA and latency conditions $L \in \{2, 4, 6, 8, 10\}$, $L \in \{2, 4, 6, 8, 10, 12, 14\}$ and $L \in \{1, 2\}$ (y-axis zoomed for visibility). (**bottom right**) Amount of architectures generated and selected in each search iteration (at most 16) that satisfy the latency constraint. In this example we searched on Edge GPU with $L = 2$.

constraint objective, for any $h \in H$:

$$\begin{aligned} &\max_{G \sim p_D} f(G) \\ &\text{s.t. } g_h(G) \leq L, \end{aligned} \tag{5.6}$$

where we drop the optimization on latency and only optimize accuracy given the latency constraint. The loss function to train our generator in these settings is updated from Equation (5.3) to:

$$\mathcal{L}(\widetilde{G}, G) = (1-\alpha)\mathcal{L}_G(\widetilde{G}, G) + \alpha\left[\lambda\mathcal{L}_{C_1}(\widetilde{G}, G) + (1-\lambda)\mathcal{L}_{C_2}(\widetilde{G}, G)\right], \tag{5.7}$$

where α is a hyperparameter trading off generation and prediction loss, and λ is a hyperparameter trading off both prediction targets C_1 (accuracy) and C_2 (latency). The risk of including multiple

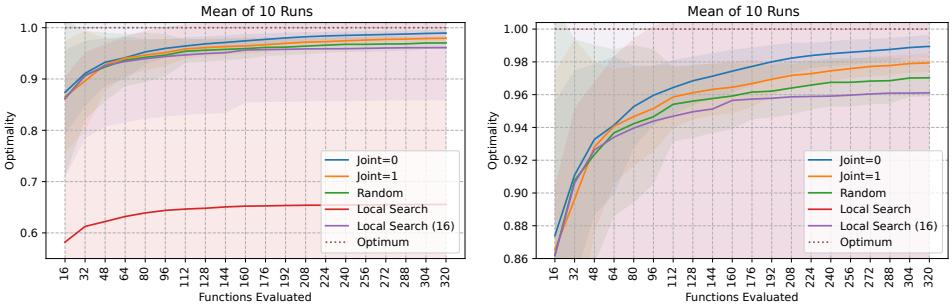


Figure 5.6: (**left**) Optimality for all search parameters in Table 5.5 at any time during the search progress in terms of the number of evaluated architectures (up to 320). Optimality is the mean validation accuracy of 10 runs per algorithm, normalized by the optimal value for each parameter setting (hence, optimum is at 1.0). (**right**) zoomed y-axis

targets to the training objective is an exploding loss leading to reduced valid generation ability of our generative network. In order to overcome this problem, we scale each loss term by the largest one, such that each term is at most 1. This way, we have a more stable training. We give a detailed overview of the hyperparameter settings in Appendix B.3. To perform LSO in the joint objective setting from Equation (5.5), we rank the training data D for both accuracy and latency jointly by summing both individual rankings. To fulfill the optimization constraint, we further penalize the ranks via a multiplicative penalty if the latency does not fulfill the constraint. This overall ranking is then used for the weight calculation in Equation (5.4). The LSO for the constraint objective setting from Equation (5.6) only ranks architectures by accuracy and penalizes architectures with infeasible latency property. We first choose random search as a baseline in this setting as it is generally regarded as a strong baseline in NAS (Li and Talwalkar, 2019). Figure 5.5 depicts searches with our model in both optimization settings on different devices (Pixel 3, Edge GPU, Raspi 4, Eyeriss, and FPGA) with different latency conditions in comparison with random search. These plots show that both methods *Joint=1* and *Joint=0* outperform the random search baseline in all different device experiments. More results are shown in Table 5.5. We observe that either optimization setting outperforms the random search baseline in almost all tasks. Additionally, our method is able to find the optimal architecture for a task regularly (in 15 out of 20 tasks), which random search was not able to provide. When considering mean accuracy and feasibility of the best architectures of all runs, we see that *Joint=1* is able to improve the ratio of feasible architectures found during the search substantially. This is to be expected given that the latent space is explicitly optimized for latency in this setting. Consequently, *Joint=1* is able to find better-performing architectures compared to *Joint=0* if the constraint restricts the space of feasible architectures strongly (see results on Raspi 4). The feasibility ratio of random search is an indicator on how restricted the space is. In most cases, the latency penalization seems to be sufficient to find enough well-performing and feasible architectures, as can be seen by the feasibility of *Joint=0* which is greatly improved compared to random search. We show the development of feasibility over time from Table 5.5 in Figure 5.5 (bottom right).

In addition to random search, local search (White et al., 2021b) is considered a strong baseline in NAS. In the case of constrained searches (as here for Hardware-Aware-NAS-Bench), we notice that it cannot perform well without adaptation. The vanilla local search algorithm expects as

input a single randomly drawn architecture from the search space. However, this architecture is not guaranteed to be feasible in this setting, as its latency can be larger than the latency constraint L. To circumvent this, we perform local search in the following settings: (a) local search vanilla setting with one randomly drawn architecture, and (b) local search initialized with 16 randomly drawn architectures. In each setting, local search continues to search the neighborhood of the next best architecture in terms of accuracy that satisfies the latency constraint. We notice that initializing local search with 16 randomly drawn architectures improves its performance substantially, however, it is still not on par with random search (Li and Talwalkar, 2019) in this constrained search space. Consequently, we only show random search as the baseline in Table 5.5 to improve readability. However, in Figure 5.6 we show the progress of our algorithms (*Joint=0* and *Joint=1*) compared to random search and local search in settings (a) and (b).

Constrained Search Algorithm

As already discussed, for this multi-objective search, we also need to modify the search process. We provide an overview of this adaption to a constrained search process in Algorithm 7.

5.3.5 *Generator Evaluation*

Here we provide additional generation ability evaluations of our generative model. Based on an investigation of autoencoder abilities from Yan et al. (2020), we can examine the generation ability of our generative model. For that we train our generator on 90% of the overall architectures in a search space, and thus have a hold-out dataset of 10% for the tabular benchmarks. The generative model training on the surrogate benchmarks is a priori only on a subset of the overall dataset. Additionally, we sample 10 000 random variables $\mathbf{z} \sim \mathcal{N}(0, 1)$ and decode them to graphs. We report the results of this investigation in Table 5.6. Here, validity describes the ratio of valid graphs our generator model generates (from this generated 10 000), uniqueness describes the portion of unique graphs from the valid generated graphs, and novelty is the portion of generated graphs *not in the training set*. It is not surprising for the NAS-Bench-301 and NAS-Bench-NLP search spaces, that our model is able to generate 100% unique and novel graphs, given the large size of both search spaces.

This demonstrates that our simple generator model is able to generate valid graphs with high novelty and consequently is able to cover a substantial part of the search space.

Table 5.6 also reports the training costs of the generative model on the complete datasets in each search space considered so far on a single Tesla V100.

5.4 ABLATION STUDIES

5.4.1 *Ablation on LSO and Backpropagation*

In this section we analyze the impact of the LSO technique and the backpropagation ability to the search efficiency. Therefore, we compare our AG-Net with the latter named adaptions on the tabular benchmarks NAS-Bench-101 and NAS-Bench-201- The results of our ablation study are reported in Table 5.7. As we can see, the lack of weighted retraining decreases the search substantially. In addition, the results without backpropagation support that the coupling of the predictor's target and the generation process enables a more efficient architecture search

Algorithm 7: Constrained Search Algorithm

Input: (i) Search space p_D
Input: (ii) Pretrained generator G
Input: (iii) Untrained performance predictor P_a
Input: (iv) Set of constraint predictors P_c
Input: (v) Query budget b
Input: (vi) e epochs to train G and P
Input: (vii) Set of constraints C

▷ `Initialize training data`
$\mathbf{D} \leftarrow \{\}$
while $|\mathbf{D}| < 16$ **do**
  $\mathbf{D} \leftarrow \mathbf{D} \cup \{d \sim p_D\}$
end
▷ `Evaluate architectures (get accuracies and constraints on target image dataset)`
$\mathbf{D} \leftarrow \text{eval}(\mathbf{D})$
▷ `Randomly initialize predictor weights`
$P_a \leftarrow \text{init}(P_a)$
foreach $P \in P_c$ **do**
  $P \leftarrow \text{init}(P)$
end
▷ `Search loop`
while $|\mathbf{D}| < b$ **do**
  ▷ `Weight train data by performance and constraints`
  $\mathbf{D}_\text{w} \leftarrow \text{weight}(\mathbf{D}, C)$
  ▷ `Train generator and predictors`
  $\text{train}(G, P_a, P_c, \mathbf{D}_\text{w}, e)$
  ▷ `Generate 100 candidates`
  $\mathbf{D}_\text{cand} \leftarrow \{\}$
  while $|\mathbf{D}_{cand}| < 100$ **do**
    $\mathbf{z} \sim \mathfrak{U}[-3, 3]$
    $\mathbf{D}_\text{cand} \leftarrow \mathbf{D}_\text{cand} \cup G(\mathbf{z})$
  end
  ▷ `Select top16 candidates with` P_a `and` P_c
  $\mathbf{D}_\text{cand} \leftarrow \text{select}(\mathbf{D}_\text{cand}, P_a, P_c, 16)$
  ▷ `Evaluate and add to data`
  $\mathbf{D} \leftarrow \mathbf{D} \cup \text{eval}(\mathbf{D}_\text{cand})$
end

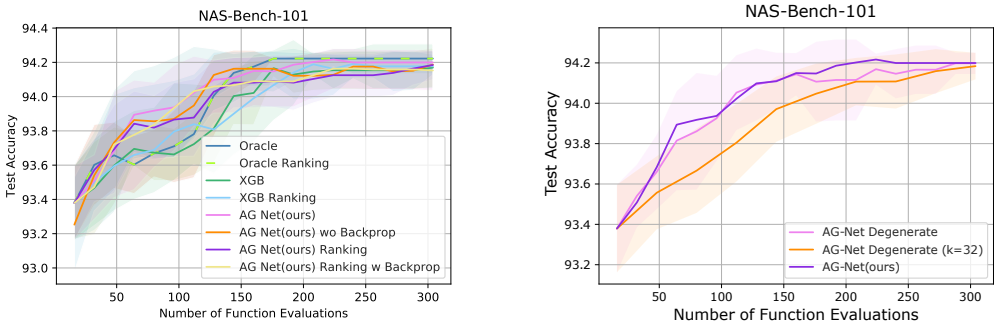
over different search spaces. Thus, the combination of LSO and a fully differentiable approach improves the effectiveness of the search the most.

Search Space	**Validity** (in %) ↑	**Uniqueness** (in %) ↑	**Novelty** (in %) ↑	**Training** (in GPU days)
NAS-Bench-101	71.69	97.92	62.30	0.4
NAS-Bench-201	99.97	73.61	10.03	0.3
NAS-Bench-301	42.27	100	100	0.9
NAS-Bench-NLP	57.95	100	100	0.7

Table 5.6: Generator Abilities and training costs. The proposed generator generates architectures with high validity and uniqueness scores. The novelty scores are in a similar range as for previous methods (Zhang et al., 2019; Yan et al., 2020).

	NAS-Bench-101		**NAS-Bench-201**					
	CIFAR-10		CIFAR-10		CIFAR-100		ImageNet16-120	
	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑
Optimum*	95.06	94.32	91.61	94.37	73.49	73.51	46.77	47.31
AG-Net (ours) w/o LSO	94.38	93.78	91.15	93.84	71.72	71.83	45.33	45.04
AG-Net (ours) w/o backprop.	94.71	94.12	**91.60**	94.30	73.38	73.22	46.62	46.13
AG-Net (ours)	**94.90**	**94.18**	**91.60**	**94.37***	**73.49***	**73.51***	**46.64**	**46.43**

Table 5.7: Ablation: Search results on NAS-Bench-101 and NAS-Bench-201 using AG-Net (mean over 10 trials with a maximal query amount of 192).



(a) Strong predictor comparison.

(b) Degenerated setting.

Figure 5.7: **(left)** Architecture search on NAS-Bench-101. Reported is the mean over 10 trials for the search of the best architecture in terms of validation accuracy on the CIFAR-10 image classification task compared to strong predictor models. **(right)** Architecture search on NAS-Bench-101 in the degenerate setting. Reported is the mean over 10 trials.

5.4.2 *Oracle Ablation*

As we have seen in the previous section, our model AG-Net is able to find high-scoring architectures in various search spaces of different sizes and with different objectives. In addition, including the supposedly stronger predictor XGB leads to improvements for the search on NAS-Bench-NLP Table 5.3. In this section, we include an even stronger architecture accuracy evaluation model,

	NAS-Bench-101		NAS-Bench-201					
	CIFAR-10		CIFAR-10		CIFAR-100		ImageNet16-120	
	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑	Val. Acc. ↑	Test Acc. ↑
Optimum*	95.06	94.32	91.61	94.37	73.49	73.51	46.77	47.31
Random Search	94.27	93.65	91.37	93.92	72.55	72.49	46.09	46.05
Local Search	94.31	93.66	91.28	94.01	72.52	72.59	45.89	46.07
Bayesian Optimization	94.27	93.62	91.30	93.99	72.23	72.35	46.09	46.01
Random Search + LSO	94.64	**94.20**	**91.61***	**94.37***	**73.49***	**73.51***	**46.77***	45.47
Local Search + LSO	94.17	93.50	91.30	93.96	72.43	72.58	45.83	45.95
Bayesian Optimization +LSO	94.50	93.96	91.43	94.17	72.64	72.67	46.30	45.91
SGNAS (Huang and Chu, 2021) + LSO	-	-	**91.61***	**94.37***	73.04	73.12	46.56	**46.32**
AG-Net (ours)	**94.96**	**94.20**	**91.61***	**94.37***	**73.49***	**73.51***	46.67	46.22

Table 5.8: Ablation: Search results on NAS-Bench-101 and NAS-Bench-201 on the AG-Net latent space (mean over 10 trials with a maximal query amount of 300).

i.e., the benchmark query input itself (oracle). The comparison of the oracle benchmark to our AG-Net and XGB modifications are visualized in Figure 5.7 (left) for NAS-Bench-101. This figure demonstrates the high performance of our model in the low query area. The more queries are evaluated for the search, the better the oracle becomes, outperforming all other methods after 150 queries.

5.4.3 *Predictor Ablation – Local Solution*

Our proposed search method in Section 5.2.6 makes the architecture search focus on promising regions in the search space. This method could be trapped in local solutions, which we investigate experimentally in the following. First, the previous section already points out that our proposed method AG-Net improves over both local search methods with and without the latent space optimization approach (Table 5.7). Thus, we assume that the latent space optimization learns properties of high-scoring architectures without being easily trapped in poor local solutions. The amount of samples drawn in each search iteration also provides a trade-off between diversity versus specificity. To investigate further how easily AG-Net could be trapped in a local solution, we test our method when it only uses in total the best k (predicted) architectures from our test samples and the training data as a new training set for the next search iteration (degenerative) and is thereby encouraged to forget about worse performing architectures. Figure 5.7 shows the search behavior of the degenerative model with $k = 16$ and $k = 32$ on NAS-Bench-101. Even in this case, AG-Net is not easily trapped in poor solutions.

5.4.4 *Latent Space Ablations*

As we have seen in Section 5.3.1, AG-Net improves over state-of-the art methods. For additional comparisons, we first investigate different baseline search methods in the latent space of the generative model, with samples z from a grid, and second, couple these baseline search methods with our LSO approach. For the first experiment we use the generator solely as a data sampler from the generator's latent space without any retraining, for the latter baseline we retrain the generator during the search. For the optimization, we use Bayesian optimization, local search and random search.

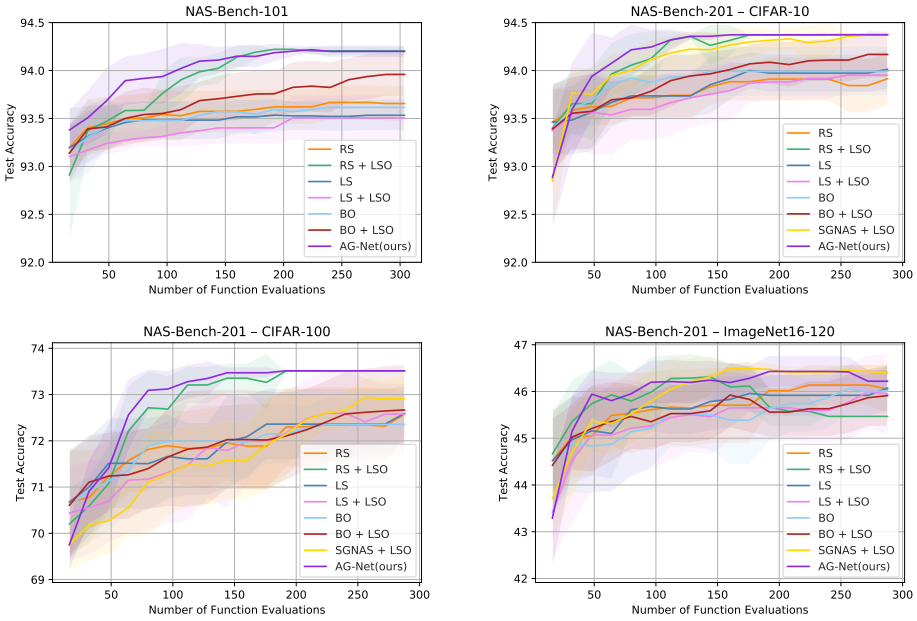


Figure 5.8: Ablation: NAS within the AG-Net generated latent space on NAS-Bench-101 and NAS-Bench-201 over 10 trials.

Bayesian Optimization We use DNGO (Snoek et al., 2015) as our uncertainty prediction model for the Bayesian optimization search strategy, with the basis regression network being a one-layer MLP with a hidden dimensionality of 128, which is trained for 100 epochs and expected improvement (EI) (Mockus, 1974) as our acquisition function, which is mostly used in NAS. We set the best function value for the EI evaluation as the highest validation accuracy of the data used to train DNGO, which is iteratively updated during the search process. We sample 16 initial random latent space variables $\mathbf{z} \sim \mathfrak{U}[-3, 3]$ and decode them to graph data using our pretrained generative model. These latent space variables and their corresponding validation architecture performances are then the inputs for the DNGO model for training. Again, the best 16 architectures are selected using EI in each round to be evaluated and added to the training data. This search ends when the total query amount of 300 is reached.

Random and Local Search In addition to BO as a comparison, we also include a random search and local search investigation. Local search evaluates samples and their neighborhood uniformly at random. An option to define the neighborhood is the set of architectures which differ from a sampled architecture by one node or edge. This can be done only in the discrete search space, given for example by the tabular NAS-Benchmarks. We have to adapt the neighborhood definition in our latent space for local search in this space. We sample a latent space variable $\mathbf{z} \sim \mathfrak{U}[-3, 3]$, decode it and evaluate the generated neural architecture. Here, we define neighborhood as the Euclidean space around the sampled latent variable $U_\epsilon(\mathbf{z}) = \{\mathbf{y} \sim \mathfrak{U}[-3, 3] | d(\mathbf{z}, \mathbf{y}) < \epsilon\}$, with ϵ being sufficiently small. This neighborhood is then investigated until a local optimum in terms of validation accuracy is reached.

Furthermore, we include a random search and local search comparison using weighted

retraining. Here, we retrain the generative model in each search iteration for 1 epoch with the weighted objective function, ceteris paribus. To compare with weight-sharing approaches, we also compare to the supernet from Huang and Chu (2021) for the NAS-Bench-201 search space. To compare our AG-Net with SGNAS, we use the supernet as our surrogate model to predict the architecture's performance while retraining the generative model in the weighted manner.

The results of our ablation studies are reported in Table 5.8. AG-Net improves over search methods on the latent space with and without LSO on both benchmarks, demonstrating that our generator in combination with our MLP surrogate model learns to adapt the distribution shift constructed by the weighted retraining best.

For further visualizations we also plot this different search methods over different query numbers in Figure 5.8 for both benchmarks NAS-Bench-101 and NAS-Bench-201. This figure demonstrates the high any-time performance of our method on both search spaces. For any number of available queries, our model is better in finding high-performing architectures from the latent space than other latent space-based methods.

5.5 Conclusion

We propose a simple architecture generative network (AG-Net), which allows us to directly generate architectures without any additional encoder or discriminator. AG-Net is fully differentiable, allowing to couple it with surrogate models for different target predictions. In contrast to former works, as also presented in Chapter 4, it enables to backpropagate the target information from the surrogate predictor into the generator. By iteratively optimizing the latent space of the generator, our model learns to focus on promising regions of the architecture space, so that it can generate high-scoring architectures directly in a query and sample-efficient manner. Extensive experiments on common NAS benchmarks demonstrate that our model outperforms state-of-the-art methods at almost any time during architecture search and achieves state-of-the-art performance on ImageNet. It also allows for multi-objective optimization on the Hardware-Aware NAS-Benchmark.

Part III

One-Shot Architecture Search

Is Differentiable Architecture Search truly a One-Shot Method?

6

So far, we investigated in part II the advantages of using generative models in NAS to improve on the search and query efficiency. In contrast, differentiable architecture search, as presented in section 2.1.4 by Liu et al. (2019), proposes a continuous relaxation of the search problem, i.e., all candidate architectures within a given search space of operations and their connectivity are jointly optimized using shared network parameters while the network also learns to weigh these operations. The final architecture can then simply be deduced by selecting the highest weighted operations.

This is appealing as practically good architectures are proposed within a single optimization run. However, as discussed in Section 2.1.4, previous works such as Zela et al. (2020a) also indicate that the proposed results are often sub-optimal, especially when the search space is not well-chosen. Specifically, since network weights are randomly initialized, promising operations can have poor initial weights such that the architecture optimization tends to entirely discard them. As a result, the practical relevance of differentiable architecture search proposed architectures depends heavily on network initialization as well as on training hyperparameters. Yet, in the context of large-scale computer vision problems such as image classification, a systematic analysis of differentiable architecture search with respect to hyperparameter optimization is hardly affordable.

In this chapter, we apply differentiable architecture search (DAS) to inverse problems with the main focus on the analysis of DAS w.r.t. the impact of domain shifts, training hyperparameter choice and network initialization. Since signal recovery has not received nearly as much attention in the NAS literature as image classification (Section 2.1), it allows to study a naive choice of parameters and settings without bias to known results and best practices. We make a clear distinction here between *DARTS* (Liu et al., 2019), which includes the proposed search space and topologies, and the differentiable architecture search, *DAS*, itself.

In the signal recovery setting, sequential architectures (Zhang et al., 2017) yield competitive results when learning to solve inverse problems, such that we can analyze the impact of the complexity of the search space more easily. Specifically, we compare the stability and sensitivity to hyperparameters of the architecture optimization in a simple, sequential search space as well as in a non-sequential search space, which we both propose, where the latter is inspired by the search space proposed in Liu et al. (2019). We investigate two types of one-dimensional inverse problems which allow for extensive experiments for each setting in order to analyze the robustness of DAS.

We show that DAS can automatically find well-performing architectures if the search space is well-preconditioned. Yet, our study also shows that the performance of DAS heavily depends on hyperparameter choices. Moreover, DAS shows a large variance for any set of hyperparameters, such that the suitability of parameters as well as the overall performance can only be judged when considering numerous runs. This finding challenges the understanding of DAS as a one-shot method for NAS. Equally concerning, we find that the estimated network performance using jointly optimized, shared weights is often not well correlated with the reconstruction ability of the final model after operation selection and re-training, i.e., the continuous relaxation in DAS seems to be quite loose. Therefore, we also meet here the problematic of *rank disorder* and *poor test generalization* (Li and Talwalkar, 2019; White et al., 2023; Zela et al., 2020a) (Section 2.1.4). In

particular, this makes the search for good hyperparameters by optimizing for the DAS training objective near-impossible. Hyperparameter optimization w.r.t. the final architecture performance is even more expensive and seems to increase the variance in the results even further. Yet, overall, our study also shows that DAS can successfully be applied to inverse problems. Specifically, it improves over the competitive random search baseline by a significant margin, when the search space contains a variety of harmful and beneficial operations. This finding is crucial, since the search space can not always be assumed to be well preconditioned in novel applications.

In summary, we make the following contributions:

- We show that applying differentiable architecture search is able to find well-performing architectures in the signal recovery setting to solve inverse problems, if the search space is well preconditioned
- We also find that DAS induces a large variance for several runs and different hyperparameter
- We find the rank disorder issue in our proposed sequential search space, and a poor test generalization in our proposed non-sequential search space.

This chapter is structured as following: In Section 6.1, we introduce the differentiable architecture search used in this chapter. Section 6.2 provides additional information about related work for reconstruction. We introduce both proposed and used search spaces in Section 6.3, which we use our experiments in Section 6.4. Lastly, we conclude this chapter in Section 6.5

This chapter a modified version of the paper J. Geiping et al. (2021a). "DARTS for Inverse Problems: a Study on Stability". In: *Advances in Neural Information Processing Systems (NeurIPS), Workshop on Deep Learning and Inverse Problems* and its extended version J. Geiping et al. (2021b). "Is Differentiable Architecture Search truly a One-Shot Method?" In: *arXiv.org* abs/ 2108.05647.

6.1 DIFFERENTIABLE ARCHITECTURE SEARCH

We introduce DAS, based on Liu et al. (2019) (Section 2.1.4), in a more detailed way in the following, as it is the basis of our analysis. While the originally proposed method optimizes so-called *cells*, which are stacked in order to define the overall neural network architecture, and defines each cell in the form of a directed acyclic graph (DAG), we conduct most parts of our systematic study of the behavior of DAS on special sequential, and easy-to-interpret meta-architectures to be described below (Fig. 6.1). To exclude that our findings are merely due to this special search space, we also consider experiments resembling the original setup in Liu et al. (2019).

Our sequential architecture consists of N nodes $x^{(i)}$, where $x^{(0)}$ represents the input data and the result $x^{(i+1)}$ of any layer is computed by applying some operation $o^{(i)}$ to the predecessor node $x^{(i)}$, i.e.,

$$x^{(i+1)} = o^{(i)}(x^{i}, \theta^{(i)}), \tag{6.1}$$

where $\theta^{(i)}$ are the (learnable) parameters of operation $o^{(i)}$. To determine which operation $o^{(i)}$ is most suitable to be applied to the feature $x^{(i)}$, one defines a set of candidate operations $o_t \in O$, $t \in \{1, \dots, |\ O\ |=: T\}$ and set the NAS optimization problem with the objective to select the optimal (discrete) arrangement of these operations in the neural architecture. DAS searches over

the continuous relaxation of this discrete problem with

$$o^{(i)} = \sum_{t=1}^{T} \beta_{o_t}^{(i)} o_t, \qquad \beta_{o_t}^{(i)} = \frac{\exp(\alpha_{o_t}^{(i)})}{\sum_{t'=1}^{T} \exp(\alpha_{o_{t'}}^{(i)})}, \tag{6.2}$$

where $\alpha = (\alpha_{o_t}^{(i)})$ are *architecture parameters* that determine the selection of exactly one candidate operation in the limit of β becoming binary. Instead of looking for binary parameters directly, the optimization is relaxed to the soft-max of continuous parameters α.

DAS formulates this search as a bilevel optimization problem in which both, the network parameters $\theta = \{\theta^{(i)}\}_{i=1}^{N}$ and the architecture parameters α, are jointly optimized on the training and validation set, respectively, via

$$\min_{\alpha} \mathcal{L}_{val}(\theta(\alpha), \alpha) \tag{6.3}$$

$$\text{s.t. } \theta(\alpha) \in \arg\min_{\theta} \mathcal{L}_{train}(\theta, \alpha), \tag{6.4}$$

where $\mathcal{L}_{val}$ and $\mathcal{L}_{train}$ denote suitable loss functions for the validation and training data. The optimization is done by approximating (6.4) by one (or zero) iterations of gradient descent, and depends on several hyperparameters such as initial learning rates, learning rate schedules and weight decays for both architecture and model parameters.

At the end of the search, the discrete architecture is obtained by choosing the most likely operation $\widehat{o}^{(i)} = \arg\max_{o_t} \alpha_{o_t}^{(i)}$ for each node. Subsequently, the final network is retrained from scratch. Thus, the fundamental assumption that justifies the idea of DAS is that the performance reached by the final network architecture on the validation set (the *architecture validation*) is highly correlated with the performance of the relaxed DAS approach obtained in Equation (6.3) (the *one-shot validation*). Only then, the architecture found during DAS optimization can also be expected to perform well after retraining.

While previous works (as introduced in Section 2.1.4) have studied a search-to-evaluation gap, i.e., the effect that the final network architecture's performance improves significantly by retraining from scratch, we further investigate whether this assumed correlation between *one-shot validation* and *architecture validation* is always given and in how far it depends on the choice of hyperparameters. This motivates our analysis of the potential benefits of DAS in a different setting than image classification.

6.2 Related Work

Previous work on reconstruction of inverse problems via learned approaches has often focused on unrolled optimization schemes, such as unrolled PDHG in Riegler et al. (2016) and Adler and Öktem (2018). These architectures, also referred to as variational networks (Klatzer et al., 2016; Hammernik et al., 2017), are constructed by unrolling existing optimization routines that solve inverse problems and adding learning components in blocks which are either recurrent, as e.g., in Aggarwal et al. (2019) or fully independent as in Hammernik et al. (2018). In this investigation we will focus on parameterized gradient descent layers which can be seen as the most fundamental building block of these optimization routines.

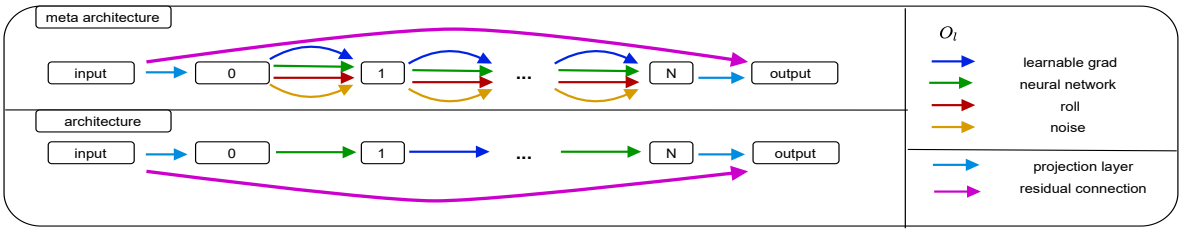


Figure 6.1: Investigated sequential meta-architecture. This setup is simple, yet it is able to represent DnCNN-like architectures (Zhang et al., 2017).

6.3 PROPOSED SEARCH SPACES

The significant advantages in computational efficiency over discrete architecture optimization methods along with the impressive performances of the final architectures have made DARTS and its variants highly attractive for automating the search for well-working neural networks. This framework itself is generic and thus applicable to any field of application, such as inverse problems.

Our following analysis of DAS for inverse problems will deliberately not be targeting settings that yield good results by design. In contrast, we propose two search spaces with different complexities that allow to analyze the stability and performance of DAS under varying degrees of difficulty, in ascending order:

- finding a good (linear) sequence of operations from meaningful choices of operations,
- finding a good (linear) sequence of operations where the set of operations to choose from contains good operations as well as harmful operations (the model needs to learn to avoid these),
- finding a good non-linear, acyclic computational graph of operations from meaningful choices of operations (this is the conventional DARTS setting),
- finding a good non-linear, acyclic computational graph of operations, where the set of operations to choose from contains good operations as well as harmful operations (the model needs to learn to avoid these).

Such search spaces allow to investigate the properties of DAS methods under various and realistic conditions. Specifically, not for all tasks, we can assume that the set of well-performing, beneficial operations is given or even complete. In such setups, one would ideally want to be able to add new operation candidates to the search space and have the search determine which configuration will work best. Therefore, it is desirable that methods perform reliable even if poor operation choices are available.

6.3.1 *Sequential Search Space*

For the simpler, sequential search space, we propose the meta-architecture shown in Figure 6.1, which should be specifically well-suited for examples of signal recovery from known data formation processes such as blurring and subsampling with noise. From a pre-defined set of operations,

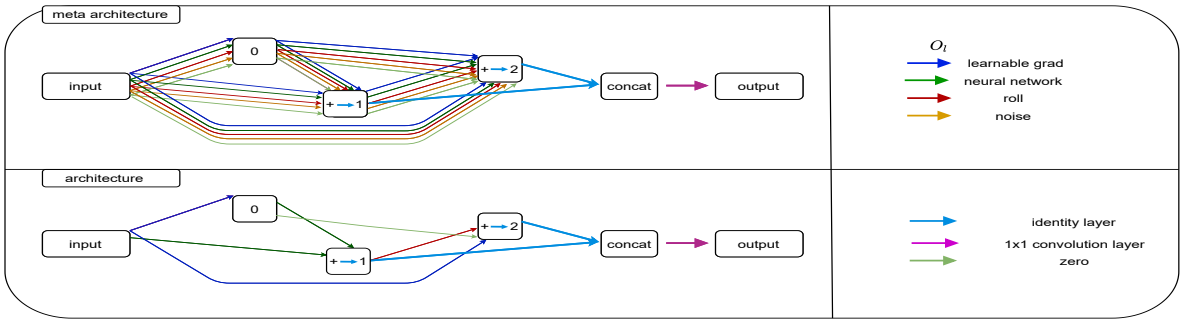


Figure 6.2: Investigated meta-architecture in the non-sequential search space.

we choose operations sequentially before adding the output to a residual branch. Image recovery networks such as DnCNN (Zhang et al., 2017) are contained in this meta-architecture. In practice, we search for 10 successive layers. A detailed discussion of the proposed operations is given in Section 6.3.3. As discussed above, the search space deliberately contains benign as well as harmful operations. This allows the evaluation of the effectiveness of DAS in any setting via the distinction of two cases: Training on all operations versus training only on beneficial operations. *A good architecture search algorithm should reliably find the optimal operations, even when presented with sub-optimal choices.*

6.3.2 *Non-Sequential Search Space*

For the more complex, non-sequential search space, we construct a cell structure with 5 states, and allow for arbitrary forward connections among the same set of operations as in the sequential setting, but also allowing for a $\{zero\}$ operation, resulting in up to 15 operational connections. The output of the last two states is then concatenated and flattened via a 1D convolution. We utilize two of these cells in succession, so that the depth of this search space with in total 10 nodes is comparable to the sequential search space from above. Figure 6.2 visualizes an exemplary cell meta architecture (supernet) during architecture search and the found cell architecture, which is then retrained. We choose in each cell one operation out of several as a connection between each node in the cell. This setting is more directly comparable to the original formulation in Liu et al. (2019), which contains a cell structure with multiple possible connections between sequential states, allowing for a larger degree of freedom in combining computational results.

The selected architecture for retraining then takes the operation between each node with the highest probability.

6.3.3 *Network Operations for Inverse Problems*

In both the sequential and non-sequential setting, we search for the optimal architecture that can be defined using operations selected from a defined set O_l. Specifically, we propose to use four operations, two of which are benign and potentially beneficial by design. The first benign operation is motivated from rolled-out-architectures (see e.g., Gregor and LeCun (2010), Schmidt and Roth (2014), and Kobler et al. (2017)) and tries to embed model-based knowledge about the recovery problems into the network's architecture. In this chapter, we consider problems which can be phrased as linear inverse problems, in which the quantity x ought to be recovered from data

$y = Ax +$ noise for a linear operator A. While the precise type of algorithm is typically dictated by (smoothness) properties of the regularization, a partially parameterized network-based approach has a lot of freedom to choose from template layers based on the mentioned inverse problem $y = Ax +$ noise

$$\arg\min_x D(Ax, y), \tag{6.5}$$

where D is a data formation term arising from the distribution of noise present, i.e.,

$$D(Ax, y) = \frac{1}{2}||Ax - y||^2$$

for Gaussian noise. This optimization objective yields templates such as a gradient descent layer:

$$x^{k+1} = x^k - \tau A^T \nabla_x D(Ax^k, y),$$

for input x^k and output x^{k+1} of a new layer. For suitable chosen τ, the application of this layer is guaranteed to reduce the objective (6.5). The gradient layer can be turned into a learnable operation by introducing a learnable mapping $\mathcal{F}(x, \theta)$ after the gradient step,

$$x^{k+1} = x^k - \tau A^T \nabla_x D(Ax^k, y) - \tau \mathcal{F}(x^k, \theta), \tag{6.6}$$

as a **learnable gradient descent layer** in our operation set O_l. The second benign layer is a fully-learned **neural network layer** in our operation set O_l, that learns an appropriate mapping $\mathcal{F}(x, \theta)$ without knowledge of the operator A:

$$x^{k+1} = \mathcal{F}\Big(x^k, \theta\Big), \tag{6.7}$$

For both layers, the learnable mapping $\mathcal{F}(x, \theta)$ is parameterized by a small convolutional network, consisting of a convolution layer, followed by batch normalization, ReLU and a second convolution layer. These two layers, learnable gradient descent layer and neural network layer, are by design beneficial operations. In contrast to these beneficial layers we also include two negative operations to each operation set; a gradient layer with white Gaussian noise, **noise layer**, and a **roll layer**, which rolls the inputs in all dimensions. In total, we set $O_l = \{\text{learnable gradient descent}, \text{ 2-layer-CNN}, \text{ roll}, \text{ noise}\}$.

6.4 EVALUATING DAS FOR INVERSE PROBLEMS

In the following, we describe the experimental setting in which we evaluate DAS for inverse problems. Thereby, we focus on small problem instances to be able to evaluate the framework not once but in several runs such as to evaluate the statistics of the results. This setup also allows to gain insights on the dependence of DAS' performance on the chosen hyperparameters.

6.4.1 *Experimental Setup*

Data Generation.

For a fast synthetic test we generate one-dimensional data sampling cosine waves of varying magnitude, amplitude and offset, and search for models to recover these samples from distorted

measurements. We consider two distortion processes with varying difficulty: First, Gaussian noise and blurring and second, in addition to these, a subsampling by a factor of 4.

We generate these synthetic one-dimensional cosine data from $N = 50$ equally spaced points ω_i on the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ with the model

$$x_i = \cos(f\omega_i + O_x) + O_y$$

for a random frequency f drawn uniformly from the interval $[0, 2\pi]$ and offsets O_x and O_y drawn from a normal Gaussian distribution. Such random drawn waves comprise our ground truth training data. We then generate measured data via the linear operation A and addition of noise η,

$$y = Ax + \eta, \qquad \eta \in \mathcal{N}(0, \sigma_\eta).$$

These pairs (y, x) represent our training data. We sample new examples on-the-fly during both training and validation, so that no confounding effects of dataset size exist. All validation and training loss evaluations are each based on 2 432 randomly drawn samples. The performance of all models is evaluated in terms of their average peak signal to noise ratio (PSNR) on validation data. The PSNR is generally defined as $10 \cdot \log_{10}(\text{MAX}^2/\text{MSE})$, where MAX^2 is the maximal value of the data, in case of image data, that would be the maximal possible image pixel value. For all experiments we chose $\sigma_\eta = 0.01$. For the *blur* experiments, the linear operator A is a Gaussian blur with kernel size 7 and $\sigma_b = 0.2$. For the *downsampling* experiments, this Gaussian blur is followed by a subsampling by a factor 4.

Hyperparameter Optimization.

Our one-dimensional case study allows us to optimize DAS training hyperparameters with more granularity than it would be possible for image classification tasks. While we run our first experiments using manually tuned hyperparameters (see Appendix B.4 for details), we also consider the behavior and stability of DAS under optimized hyperparameters. We stress that we consider this mainly as an analysis tool - given that NAS itself is a hyperparameter optimization on which we stack another, and acknowledge that this optimization is practically intractable when larger problems are considered. To improve hyperparameters, we apply BOHB (Falkner et al., 2018), a Bayesian optimization method with hyperband (Li et al., 2018a) and run BOHB for 128 hyperband iterations, which is an affordable budget in this one-dimensional data setting. It is important to note that BOHB is not an exhaustive search and thus there are no guarantees for success within our budget or even in general, in a way that it finds the globally optimal hyperparameters for the just mentioned objective. The usage of BOHB as such covers the problem of hyperparameter optimization partially, but in general, there is no simple fix of DAS via hyperparameter search - which is itself a notable statement about the algorithm. In particular, we optimize the hyperparameters with respect to first the one-shot validation performance, “BOHB-one-shot”, and second the final architecture performance, “BOHB”. Note here that hyperparameter search that maximizes the final architecture performance instead of the one-shot validation performance is twice as expensive (on top of the already expensive hyperparameter search) due to the need for retraining.

6.4.2 *Results*

We first investigate the performance of DAS on the simplified sequential search space presented in Section 6.3.1 to validate the one-shot search property of DAS. For our analysis, we are not

Operations	**Method**	**Architecture Val.** (PSNR) (↑)						**Runtime**
		Blur			**Downsampling**			
		Max.	Mean	Med.	Max.	Mean	Med.	sec.
Good ops.	Learnable Grad. only	17.45	16.36	16.49	14.35	13.24	13.55	0:57
	Nets only	21.63	19.45	20.71	16.92	13.13	14.05	0:57
	DAS	23.46	21.56	21.60	18.03	16.36	16.66	2:39
	Random Sampling	24.04	22.00	22.16	18.10	16.74	16.78	0:57
	Random Search	**24.04**	**22.85**	**22.75**	**18.10**	**17.62**	**17.56**	2:55
All	DAS	**22.86**	**15.64**	**18.57**	**18.01**	**15.39**	**16.12**	2:53
	Random Sampling	20.86	9.45	8.10	13.78	5.08	4.31	0:57
	Random Search	20.86	12.39	12.34	13.78	9.61	9.77	2:55

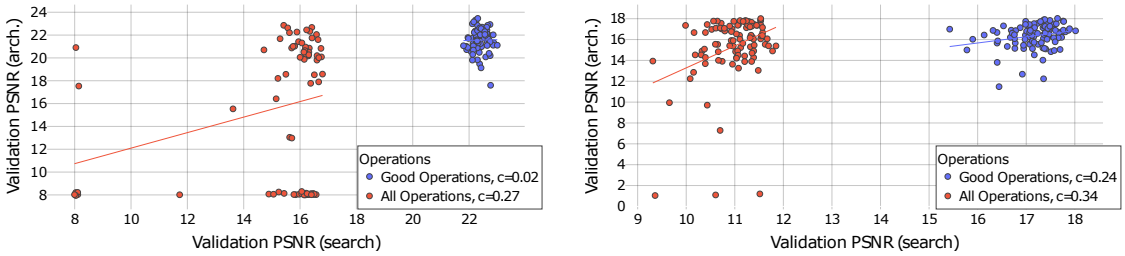
Table 6.1: Architecture validation PSNR values for 1D inverse problems. Shown is the maximal, mean and median PSNR over 75 trials in the sequential search space.

only interested in the best found architecture but also in the statistics of the search to leverage the advantages of the proposed efficient setup. We therefore evaluate 75 trials of DAS as well as several baselines such as (i) setting all operations to *Learnable Grad.* or *Net* (i.e., learnable gradient descent or 2-layer CNN), (ii) picking a random architecture (random sampling), and (iii) performing a random search within an equal time budget as required by one DAS run. We summarize the results in Table 6.1. We furthermore distinguish between different operation sets: only good operations (good ops.) and the complete operation set O_l (all ops.).

The first results for DAS indicate that it works well for inverse problems. It proposes successful architectures given the complete operation set for both considered data formation methods, *blur* and *downsampling*, resulting in architectures with a median PSNR of 18.57 and 16.12. Thereby, it also outperforms architectures consisting of only one good operation in both operations set cases, especially when considering the best found architecture using DAS. Practically, these experiments thus lead to a first interesting result for applied inverse problems: The best found architecture is a hybrid version that mixes both beneficial operations, possibly suggesting that the best way to approach inverse problems are neither plain (convolutional) networks nor pure unrolling schemes.

Next, we compare DAS to random sampling (random selection of the operation at each layer) and random search approaches. To allow for comparison at an equal time budget for the latter (random search in Table 6.1), we evaluate 5 randomly sampled architectures and report the best for each trial. One random evaluation using only good operations, i.e., the training of one random architecture, takes on average 57 sec. versus 2 min. 39 sec. for one DAS run. For the sequential search space that purely consists of benign operations, random search outperforms DAS with a median PSNR of 22.75 versus 21.6 on *blur* and 17.56 versus 16.66 on *downsampling*. Thus, in this scenario, *random search outperforms DAS when used as a one-shot model.* In addition, the simple random sampling approach also outperforms DAS. This is different when harmful operations are added. For a search on the full operations set O_l, DAS can clearly outperform both simple random baselines.

For further analysis, we additionally investigate how many random search runs are needed, to improve over the DAS median for the set of all operations, i.e., 18.57 PSNR on blur: Random search needs on average (of 10 runs) 49 random sampling steps to improve over the DAS median. While



(a) Blur. (b) Downsampling.

Figure 6.3: Scatter plot corresponding to Table 6.1 showing architecture PSNR (y-axis) plotted against 1-shot validation PSNR (i.e., the validation performance on the DAS objective) for blur (**left**) and downsampling (**right**).

this observation is overall motivating, we also observe that the performance of DAS significantly drops on average as well as in the median when all operations are considered (compared to only using benign operations). Especially on the blurred data the PSNR drops in the median from 21.60 to 18.57. This effect is undesired: ideally, DAS should be able to reliably filter out harmful operations.

We investigate the performance of DAS in the non-sequential search space in Section 6.4.5, which shows lower performance than in the sequential search space, and thus we consider only the latter one in the following investigations. From a theoretical perspective, we argue that DAS should be able to determine which operations are harmful: If we assume that the validation accuracy during the optimization correlates with the validation accuracy of the final architecture, harmful operations should be excluded early on in the optimization process. Therefore, in the following section, we study this correlation and investigate whether the behavior can be improved by optimizing training hyperparameters.

6.4.3 *Architecture and DAS Performance*

Figure 6.3 takes a closer look at the trials considered in Table 6.1, scattering the values of all trials separately with architecture performance (y-axis), which is computed after retraining the final architecture versus the direct validation performance of the one-shot architecture (x-axis). We also plot a regression line over all trials and report the correlation of all trials in the legend, showing the linear fit has limited expressiveness. As discussed, the correlation of these quantities is a fundamental assumption of DAS. However, this first experiment indicates a correlation problem: The assumption that a better *one-shot validation* implies a better *architecture validation* does not always seem to be true. Therefore, we are faced with the problem of rank disorder here as well, as introduced in Section 2.1.4.

These plots also show that DAS' behavior is highly problem-dependent: The *downsampling* dataset (right), shows that, although the mean value of DAS can be non-optimal, search performance and architecture performance are weakly correlated, even if the best architecture only has average search performance. The closely related *blur* dataset (left) shows an entirely different behavior with different "failure" cases, from which we can observe with the given hyperparameters that 1) either DAS proposes architectures with low (one-shot) search validation PSNR (i.e., it

Data	Hyperparameters	Architecture Val. (PSNR) (↑)					
		Good Ops.			All		
		Max.	Mean	Med.	Max.	Mean	Med.
Blur	H1	23.46	21.56	21.60	22.86	15.64	18.57
	H2	23.46	21.43	21.63	**23.10**	**16.77**	**19.88**
	BOHB-one-shot-Blur	22.83	20.86	20.75	22.47	15.57	18.04
	BOHB-one-shot-DS	22.33	20.65	20.96	22.41	14.43	14.41
	BOHB-Blur	**23.57**	**22.05**	**22.38**	22.94	12.76	8.21
Downsampling	H1	18.03	16.36	16.66	18.01	15.39	16.12
	H2	18.20	16.57	16.78	17.82	**15.93**	**16.21**
	BOHB-one-shot-Blur	**18.42**	**16.83**	**16.95**	17.73	14.36	14.57
	BOHB-one-shot-DS	17.51	15.33	15.84	**18.12**	12.36	13.65
	BOHB-Blur	18.26	14.63	15.93	17.91	15.04	15.44

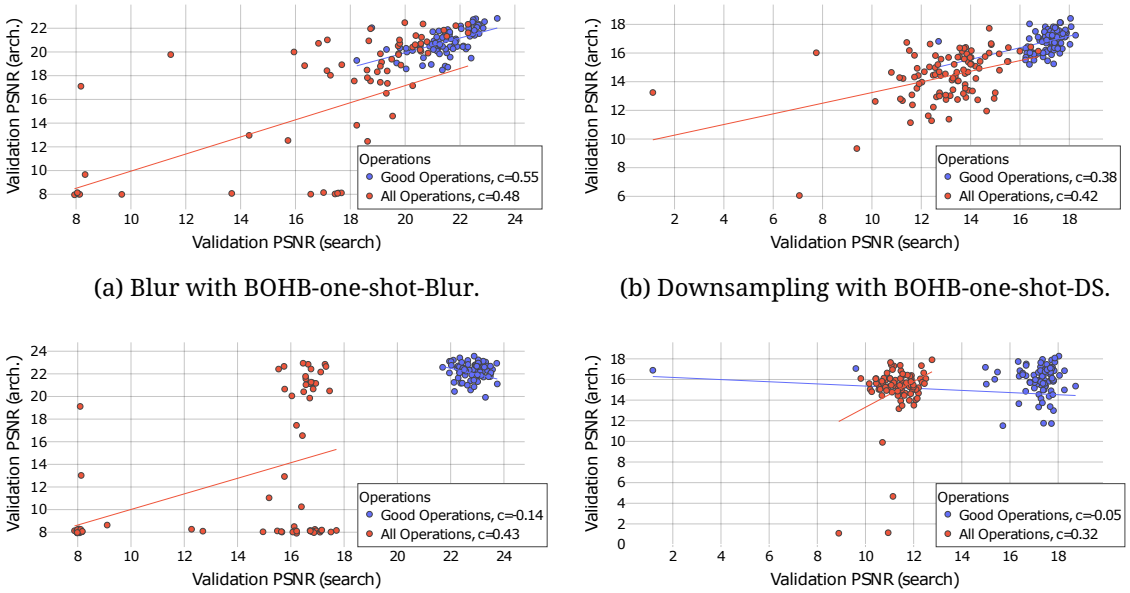
Table 6.2: Architecture validation PSNR values for 1D inverse problems with different hyperparameter. Shown is the maximal, mean and median PSNR over 75 trials.

fails), or that 2) DAS works but does not predict a useful architecture (low architecture validation PSNR), or that 3) DAS does predict a useful architecture but is unrelated to its search performance. Only the best proposed architectures perform well in both. To further analyze the correlation, we investigate DAS behavior with different training hyperparameters.

We evaluate DAS using 5 different training hyperparameter sets; two are chosen manually, H1 and H2 (H1 are the hyperparameters used in the previous Section 6.4.2), whereas the other three are tuned using BOHB, as described in Section 6.4.1. We use BOHB to tune hyperparameters for the one-shot validation performance for both *blur* (BOHB-one-shot-Blur) and *downsampling* (BOHB-one-shot-DS), individually, and also to target the final validation performance for *blur* (BOHB-Blur). The DAS search results for different training hyperparameters are given in Table 6.2. For additional visualization, we plot the results for all BOHB found hyperparameter trials in Figure 6.4. The plot shows that the correlation for both data formation methods increases with the corresponding BOHB-one-shot tuned hyperparameters, with also a higher range of the search validation PSNR.

This experiment also shows a rather surprising outcome: In the case of *blur*, the average performance (using the dataset associated BOHB fine-tuned hyperparameters) is on par with the manually chosen hyperparameters H1 and H2, whereas the performance for *downsampling* decreases, especially when all operations are considered. In addition, the best architecture PSNR over 75 trials decreases. Overall, the apparent stabilization via optimization of the search loss removes not only negative, but also positive outliers. Furthermore, hyperparameters optimized for one dataset do not transfer well to the other.

Using BOHB to target the final validation performance for *blur* (BOHB-Blur) instead of the one-shot validation performance has also a positive impact on the one-shot validation and architecture validation correlation (Figure 6.4), compared to the manually chosen hyperparameters H1 and H2 in Figure 6.3, but not to the same amount as for the BOHB-one-shot hyperparameters. However, these hyperparameters successfully increase the max. architecture performance. Overall, hyperparameters optimized with BOHB on the one-shot validation have to be considered with caution. This can be seen by cross-checking their performance, i.e., evaluating the BOHB-one-



(a) Blur with BOHB-one-shot-Blur.

(b) Downsampling with BOHB-one-shot-DS.

(c) Blur with BOHB-Blur.

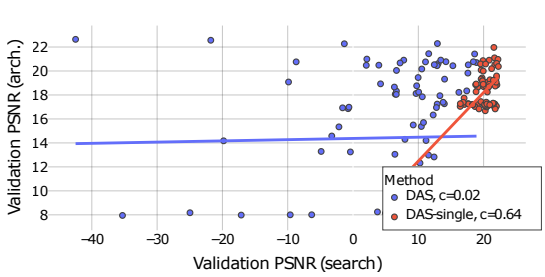
(d) Downsampling with BOHB-Blur.

Figure 6.4: Scatter plot corresponding to Table 6.2 with BOHB-optimized hyperparameters, showing architecture PSNR (y-axis) plotted against one-shot validation PSNR (x-axis). **(top left)** Blur with hyperparameters *BOHB-one-shot-Blur*. **(top right)** Downsampling with hyperparameters *BOHB-one-shot-DS*. **(bottom left)** Blur with hyperparameters *BOHB-Blur*. **(top right)** Downsampling with hyperparameters *BOHB-Blur*.

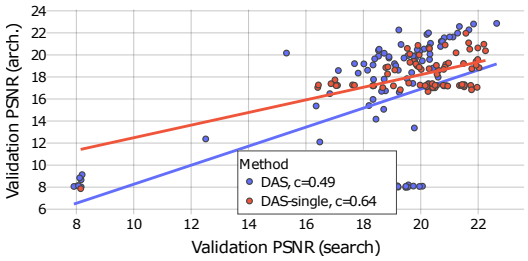
shot-Blur hyperparameters for *downsampling* and the BOHB-one-shot-DS hyperparameters for *blur*. For the case of *blur* and *all operations* in Table 6.2, the dedicated BOHB-one-shot-Blur hyperparameters are significantly more stable (measuring median PSNR) than the BOHB-one-shot-DS hyperparameters, although their maximal PSNR is very close.

When changing the domain to *downsampling*, the exact opposite holds: BOHB-one-shot-Blur hyperparameters improve over BOHB-one-shot-DS hyperparameters in terms of stability. Note that this could be due to both, the missing correlation between one-shot and architecture validation as well as the missing guarantee of any Bayesian search to find the optimal hyperparameters. In addition, Table 6.2 even demonstrates that the manual hyperparameters H1 and H2 lead to a better average performance compared with dedicated BOHB tuned hyperparameters, especially in the case of only good operations.

In conclusion, we find two schools of thought when evaluating the performance of DAS. For maximal performance, we should *not understand DAS as a one-shot search approach*, but as a component in a *larger search that proposes trial architectures*. For average performance, and immediate performance with a single DAS run, we should be optimizing the search performance and maximize its correlation with architecture performance - although as our experiments show, this is non-trivial even when searching for these hyperparameters in an automated fashion. We stress that the two directions are not at odds with each other, yet problems can arise in the literature when comparing proposed improvements of DAS across both. Some algorithmic improvements of DAS are more likely to improve best-case performance, whereas others are more



(a) Blur with BOHB-one-shot-DAS-single.



(b) Blur with respectively BOHB-one-shot optimized hyperparameters.

Figure 6.5: **(left)** Architecture search with BOHB-searched hyperparameters for DAS with single level optimization on the blur data formation. **(right)** Both methods with their own BOHB hyperparameters on the blur data formation.

likely to impact single trial stability. If these two are not carefully compared, then best-case results, which do provide better benchmark numbers, can appear to supersede stability results. Here, we discuss this effect for a simplified case study, but for large-scale DAS in image classification, where trials are expensive and fixed random seeds tempting, such a dichotomy makes it fairly difficult to evaluate and classify the manifold improvements of DAS.

6.4.4 *Improving the Initialization*

Several works, such as Zela et al. (2020a), investigate the instability of the bilevel approximation of DAS w.r.t. the weight initialization; the random initialization of the network weights can cause promising operations having poor initialization and thus tend to be entirely discarded during the architecture search. We evaluate the impact of this initialization by modifying the DAS search, such that it only has to search for the optimal architecture parameters to build the resulting architecture. In this section, we only consider the *blur* data formation and search on the complete operation set O_l.

For this *DAS-single* approach, we pretrain the operations {learnable gradient descent} and {2-layer-CNN} as baseline architectures, compared to *Learnable Grad. only* and *Nets only* from Section 6.4.2, and keep the operation weights fixed. This is generally only possible for the feed-forward architectures that we consider and requires only a weak specialization between layers. Thereby, we avoid the random initialization of the operations weights for the DAS search. Figure 6.5 shows the results of DAS-single search with BOHB-optimized hyperparameters for the complete operation set. Notably, BOHB-optimized hyperparameters for the DAS-single one-shot validation (Figure 6.5 left) lead to a positive impact on the correlation of the one-shot and architecture validation PSNR using DAS-single and to a negative impact for DAS. In addition, when comparing DAS and DAS-single with their hyperparameters being individually optimized with BOHB with respect to their one-shot validation (Figure 6.5 right), DAS finds a higher architecture validation PSNR than DAS-single, whereas DAS-single becomes more robust against possible outliers, making this search less sensitive.

Operations	Method	Architecture Val. (PSNR) (↑)					
		Blur			Downsampling		
		Max.	Mean	Med.	Max.	Mean	Med.
Good ops.	Learnable Grad. only	13.19	12.41	12.38	11.30	8.89	9.59
	Nets only	**16.35**	**14.83**	**15.82**	**13.63**	**13.07**	**13.06**
	DAS	15.34	13.08	12.51	13.22	10.22	10.43
	Random Sampling	16.20	11.29	11.88	13.15	6.26	5.72
All	DAS	**16.15**	**13.56**	**13.73**	13.31	**9.47**	**8.61**
	Random Sampling	16.05	9.56	8.17	**13.39**	4.37	3.21

Table 6.3: Architecture validation PSNR values for 1D inverse problems for the non-sequential search space. Shown is the maximal, mean and median PSNR over 100 trials.

6.4.5 *Results Non-sequential Search Space*

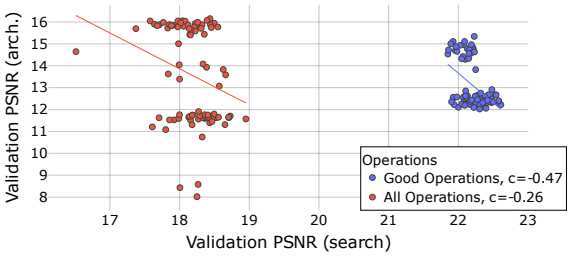
Since the original formulation in Liu et al. (2019) contains a cell structure with multiple possible connections between sequential states, allowing for a larger degree of freedom in combining computational results, it is a priori conceivable that some stability of DAS could be conferred through this structure. Therefore, we now analyze the DAS performance on the non-sequential DAS-like search space as presented in Section 6.3.2 and exemplified in Figure 6.2. Table 6.3 shows the results and that this wider search space does not improve the overall performance in comparison to Table 6.1. Indeed the non-sequential search space hampers not only the DAS search significantly but also all other baseline approaches, resulting in lower architecture performances for both data formations. In this setup, the *Nets only* baseline, which uses the 2-layer CNN for all possible operations, performs best.

As in Section 6.4.2, we observe a significant drop in the performance of DAS for *downsampling* when harmful operations are included in the search space. In this case of all operations, DAS can significantly outperform the random sampling baseline but not reliably determine the obviously best operation (neither using all ops. nor using only good ops.).

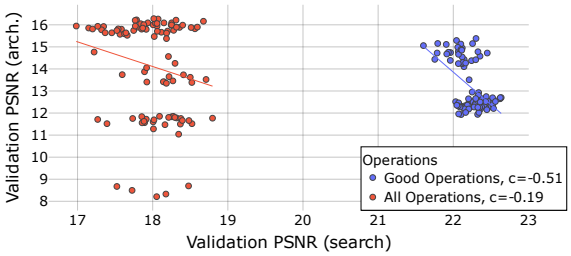
Hyperparameters

In this section, we additionally investigate the stability of our DAS framework with respect to hyperparameters within the non-sequential search space from Section 6.3.2 for the *blur* data formation.

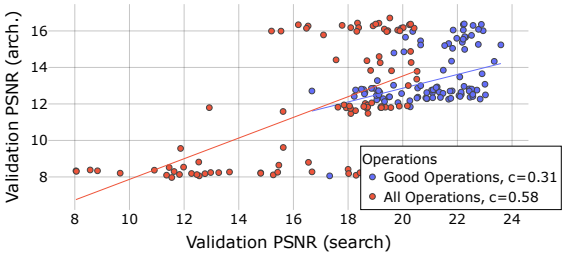
To investigate the hyperparameter stability further for this non-sequential search space, we conduct experiments using the same BOHB-optimized hyperparameters as in Section 6.4.3 (Figure 6.6) and additionally included BOHB-optimized hyperparameters for this non-sequential search space for first targeting the one-shot validation performance (BOHB-Non-Seq-one-shot-Blur) and second targeting the final architecture performance (BOHB-Non-Seq-Blur) (Figure 6.7). Table 6.4 shows the overall results, which are similar to Section 6.4.3: changing the hyperparameters in this non-sequential search space does not improve the stability of the search process. Figure 6.6 shows all trials for the non-sequential search space for all hyperparameters from the sequential search space *H1* (top left) and *H2* (top right). These plots clarify further, that the search space change does not improve the DAS search process. The correlation between the one-shot validation and the architecture validation even becomes negative. Yet, these plots also show 2



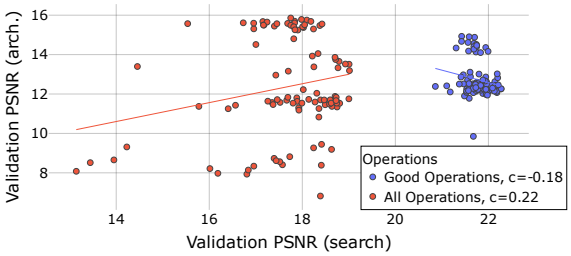
(a) Non-Sequential Blur with H1.



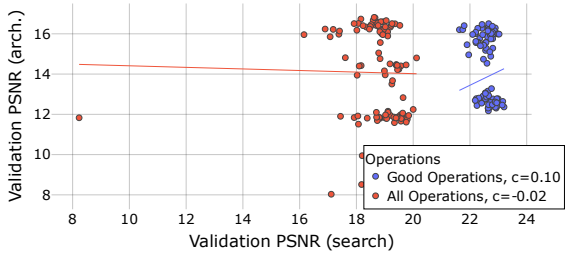
(b) Non-Sequential Blur with H2.



(c) Non-Sequential Blur with BOHB-one-shot-Blur.



(d) Non-Sequential Blur with BOHB-one-shot-DS.

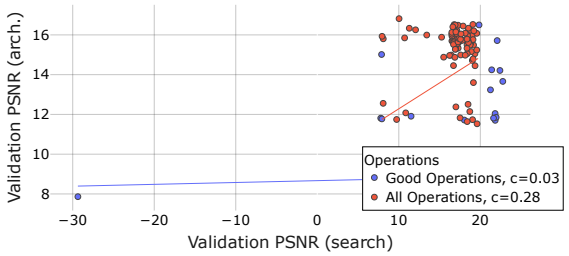


(e) Non-Sequential Blur with BOHB-Blur.

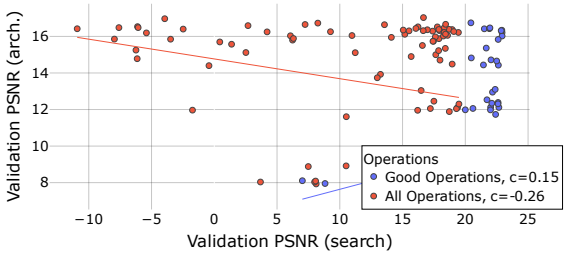
Figure 6.6: Scatter plot for the non-sequential DAS search space on *blur* corresponding to Table 6.4 showing architecture PSNR (y-axis) plotted against one-shot validation PSNR (x-axis). (**top left**) Hyperparameters *H1*. (**top right**) Hyperparameters *H2*. (**middle left**) Hyperparameters *BOHB-one-shot-Blur*. (**middle right**) Hyperparameters *BOHB-one-shot-DS*. (**bottom**) Hyperparameters *BOHB-Blur*.

different "failure" cases for both operations sets, only beneficial operations and all operations, and both data formations: The validation PSNR is stable, whereas the architecture validation performance is clustered in two different regions, one being very low and the other being around 15 PSNR. Note, the mean architecture validation PSNR for all operations in the sequential search space from Section 6.4.3 in Table 6.2 is also around 15 PSNR.

For additional visualization, we also display the results using BOHB found hyperparameters in the sequential search space in Figure 6.6 (*BOHB-one-shot-Blur* (middle left), *BOHB-one-shot-DS* (middle right), and *BOHB-Blur* (bottom)), as well as BOHB found hyperparameters tuned for this non-sequential search space in Figure 6.7. However, hyperparameter search for the non-sequential search space via BOHB on both the one-shot validation performance and the architecture performance as a target, does not actually improve the stability of the search for this



(a) Non-Sequential Blur with BOHB-Non-Seq-one-shot-Blur.



(b) Non-Sequential Blur with BOHB-Non-Seq-Blur.

Figure 6.7: Scatter plot for the non-sequential DAS search space on blur with hyperparameters searched for this search space, showing architecture PSNR (y-axis) plotted against one-shot validation PSNR (x-axis). (**left**) Blur with hyperparameters *BOHB-Non-Seq-one-shot-Blur*. (**right**) Blur with hyperparameters searched for the final architecture performance *BOHB-Non-Seq-Blur*.

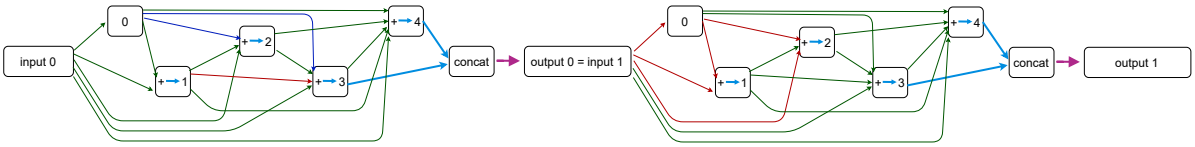
Data	**Hyperparameters**	**Architecture Val.** (PSNR) (↑)					
		Good Ops.			**All**		
		Max.	Mean	Med.	Max.	Mean	Med.
Blur	H1	15.34	13.08	12.51	16.15	13.56	13.73
	H2	15.38	13.17	12.52	16.28	**14.11**	**15.58**
	BOHB-one-shot-Blur	16.38	13.25	12.76	16.71	11.73	11.8
	BOHB-one-shot-DS	14.93	12.73	12.44	15.86	12.37	11.72
	BOHB-Blur	16.50	**13.83**	**13.06**	16.82	14.07	14.44
	BOHB-Non-Seq-one-shot-Blur	16.50	8.84	8.09	16.82	13.96	15.5
	BOHB-Non-Seq-Blur	**16.74**	9.73	8.11	**17.03**	13.45	15.42

Table 6.4: Architecture validation PSNR values in the non-sequential search space for 1D inverse problems with cosine data. Shown is the maximal, mean and median PSNR over 75 trials.

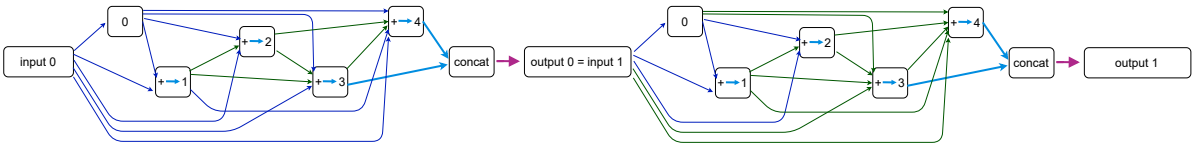
new search space, as demonstrated in Figure 6.7. Accordingly we find on the one hand that the findings in the previous Section 6.4.2 regarding non-applicability of DAS as a one-shot model for inverse problems translate to a cell-based search space and on the other hand (investigating the overall performance metrics for both search spaces), that the sequential search space appears to be a helpful prior for architecture search for inverse problems, given that its PSNR scores are overall higher. Concluding, this non-sequential search space shows not only the problem of rank disorder as for the sequential search space but also a poor test generalization, which is a common problem in differentiable architecture search (see Section 2.1.4).

Visualizations

Here we visualize in Figure 6.8 two found architectures using the H1 hyperparameters for the operation sets “all operations” and “only good operations” for the data formation blur in the non-sequential search space from the experiments above.



(a) All operations.



(b) Good operations.

Figure 6.8: Found architectures in the non-sequential search space for two different operation sets for the data formation blur. Hyperparameter H1 is used for these searches for all operations (**top**) and only beneficial operations (**bottom**).

6.5 CONCLUSIONS

In this chapter we analyze DAS in a systematic study on one-dimensional inverse reconstruction problems. In this setting, we show that DAS improves over a random search baseline by a significant margin, if the available set of beneficial operation is not determined in advance. In our analysis, we make the following findings: While it is possible to find well-performing architectures using DAS, multiple runs of the same setting yield a high variance, challenging the common understanding of DAS as a one-shot method. Moreover, the ability to find well-performing architectures is highly dependent on the specific choice of hyperparameters. Unfortunately, judging the success of any DAS-based model right after the one-shot training is difficult, since a strong correlation to the actual architecture performance is missing. As such, even automatic hyperparameter searches such as BOHB cannot faithfully be applied to the one-shot loss. Therefore, we emphasize for the future the necessity to

1. look at a full statistical evaluation of DAS performances over multiple trials, in all applications where this is feasible,

2. and show a reasonable correlation between the search and final architecture performances for any method that reports improved results based on a more faithful minimization of the one-shot DAS objective.

Part IV

Robustness of NAS Architectures

A Dataset for Neural Architecture Design and Robustness

7

Innovations in the architecture design led to an ever-improving performance of deep neural networks, which resulted in the research direction *Neural Architecture Search*. The goal of NAS is to automatically find architectures with high scores. So far, we presented search approaches (Part II) that successfully tackled exactly this goal and resulted in efficient search approaches to find high-scoring architectures.

However, the goal of new architectures and their design is not only high performance but the increased emphasis is also placed on the robustness of the networks, especially in computer vision. This led to a shift also in NAS research in which the search for new architecture designs with ever-better performance is accompanied by the search for architectures that are robust against adversarial attacks and corruptions. This is important, since image classification networks can be easily fooled by adversarial attacks crafted by already light perturbations on the image data, which are invisible for humans. This leads to false predictions of the neural network with high confidence.

Robustness in NAS research combines the objective high-performing and robust architectures (Devaguptapu et al., 2021; Dong et al., 2020a; Hosseini et al., 2021; Mok et al., 2021). However, there was no attempt so far to evaluate a full search space on robustness, but rather search for architectures in the wild using one-shot methods. In this chapter, we present a first step towards closing this gap. We are the first to introduce *a robustness dataset* based on evaluating a *complete* NAS search space, such as to allow benchmarking NAS approaches for the robustness of the found architectures. This will facilitate better streamlined research on neural architecture design choices and their robustness. We evaluate all 6 466 unique pretrained architectures from the NAS-Bench-201 benchmark (Dong and Yang, 2020) on common adversarial attacks (Goodfellow et al., 2015; Kurakin et al., 2017; Croce and Hein, 2020) and corruption types (Hendrycks and Dietterich, 2019). We thereby follow the argumentation in NAS research that employing one common training scheme for the entire search space will allow for comparability between architectures. Having the combination of pretrained models and the evaluation results in our dataset at hand, we further provide the evaluation of common training-free robustness measurements, such as the Frobenius norm of the Jacobian matrix (Hoffman et al., 2019) and the largest eigenvalue of the Hessian matrix (Zhao et al., 2020), on the full architecture search space and use these measurements as a method to find the supposedly most robust architecture.

To prove the promise of our dataset to promote research in NAS for robust models we perform several common NAS algorithms on the clean as well as on the robust accuracy of different image classification tasks. Additionally, we conduct a first analysis of how certain architectural design choices affect robustness with the potential of doubling the robustness of networks with the same number of parameters. This is only possible, since we evaluate the whole search space of NAS-Bench-201, enabling us to investigate the effect of small architectural changes.

To our knowledge we are the first to introduce a robustness dataset covering a full (widely used) search space allowing to track the outcome of fine-grained architectural changes. In summary, we make the following contributions:

- We present the first robustness dataset evaluating a complete NAS architectural search

space on robustness.

- We present different use cases for this dataset; from training-free measurements for robustness to neural architecture search.
- Lastly, our dataset shows that a model's robustness against corruptions and adversarial attacks is highly sensitive towards the architectural design, and carefully crafting architectures can substantially improve their robustness.

This chapter introduces in Section 7.1 robustness in a general way and provides in Section 7.2 relevant related work on corruptions and adversarial attacks, as well as robustness in NAS. We present the dataset generation in Section 7.3 and show first use cases in Section 7.4. Further, Section 7.4.3 provides first analysis of how the architecture design and topology influences the robustness. We conclude this chapter in Section 7.5.

This chapter is a slightly modified version of the paper S. Jung et al. (2023). "Neural Architecture Design and Robustness: A Dataset". In: *Proc. of the International Conference on Learning Representations (ICLR)*. The code for this chapter is available on GitHub[1].

7.1 ROBUSTNESS AND GENERALIZATION

Although neural networks achieve unprecedented results on image classification tasks (He et al., 2015), even surpassing human performance, they seem to be easily fooled by small perturbations and struggle to generalize to out-of-distribution data. In the context of the former the goal of a so-called adversarial attack is to cause a false prediction of networks with high confidence using perturbations barely visible for humans. Formally, following Croce and Hein (2020), let the C-class neural network be given by $f_\theta : \mathbb{R}^D \to \mathbb{R}^C$, parameterized by network weights θ, with input point $x \in \mathbb{R}^D$ and the corresponding true label $y \in \mathbb{R}^C$. The feasible set of adversarial attacks is further defined as $\{x' \in \mathbb{R}^D | d(x, x') \leq \epsilon\}$, with $d(x, x') := \|x - x'\|_p$, $p \in \{2, \infty\}$, and $\epsilon > 0$. The most popular attacks use these L_p-distances. The sample x' is an adversarial sample for x if it successfully changes the predicted label from the true label to a wrong label. An adversarial attack x' can be found by solving the optimization problem in terms of the loss function $\mathcal{L}$

$$\begin{aligned} &\max_{x' \in \mathbb{R}^D} \mathcal{L}(f_\theta(x'), y) \\ &\text{s.t. } d(x, x') \leq \epsilon \end{aligned} \tag{7.1}$$

We differentiate here between white-box and black-box attacks; for the former one assumes full access to the model, whereas the latter only has limited access to the model, such as only the model's prediction (Croce and Hein, 2020). We will provide more information for both types of adversarial attacks in Section 7.3.2.

However, adversarial attacks are only one part to determine the robustness of an architecture. In addition, neural networks often also fail to generalize under various image corruptions, which change the image data distribution, such as noise, blurring, and different weather conditions (Hendrycks and Dietterich, 2019) as we will discuss further in Section 7.2 and Section 7.3.3. In this context of low generalizability, Geirhos et al. (2019) analyze the prediction behavior of popular

[1] https://github.com/steffen-jung/robustness-dataset

CNNs and observe a so-called texture-shape cue conflict, i.e., they are biased towards texture recognition (texture bias), which is in contrast to human vision behavior that focuses on shape information (shape bias). In addition, they observe that increasing the shape-bias comes with the benefit of additional robustness and better generalization behavior.

7.2 Related Work

Common Corruptions

As discussed, common corruptions such as Gaussian noise or blur can cause the performance of neural architectures to degrade substantially (Dodge and Karam, 2017). For this reason, Hendrycks and Dietterich (2019) propose a benchmark that enables researchers to evaluate their network design on several common corruption types.

Adversarial Attacks

Szegedy et al. (2014) showed that image classification networks can be fooled by crafting image perturbations (adversarial attacks as presented in Section 7.1) that maximize the networks' prediction towards a class different to the image label. Surprisingly, these perturbations can be small enough such that they are not visible to the human eye. One of the first adversarial attacks, called fast gradient sign method (FGSM) (Goodfellow et al., 2015), tries to flip the label of an image in a single perturbation step of limited size. This is achieved by maximizing the loss of the network and requires access to its gradients. Later gradient-based methods, like projected gradient descent (PGD) (Kurakin et al., 2017), iteratively perturb the image in multiple gradient steps. To evaluate robustness in a structured manner, Croce and Hein (2020) propose an ensemble of different attacks, including an adaptive version of PGD (APGD) (Croce and Hein, 2020) and a black-box attack called Square Attack (Andriushchenko et al., 2020) that has no access to network gradients. Croce et al. (2021) conclude the next step in robustness research by providing an adversarial robustness benchmark, RobustBench, tracking state-of-the-art models in adversarial robustness.

Robustness in NAS

With the increasing interest in NAS in general, the aspect of robustness of the optimized architectures has become more and more relevant. Devaguptapu et al. (2021) provide a large-scale study that investigates how robust architectures found by several NAS methods, such as Liu et al. (2019), Cai et al. (2019), and Xu et al. (2020), are against several adversarial attacks. They show that these architectures are vulnerable to various different adversarial attacks. Guo et al. (2020a) first search directly for a robust neural architecture using one-shot NAS and discover a family of robust architectures. Dong et al. (2020a) constrain the architectures' parameters within a supernet to reduce the Lipschitz constant and therefore increase the resulting networks' robustness. Few prior works such as Carlini et al. (2019), Xie et al. (2019a), Pang et al. (2021), and Xie et al. (2020) propose more in-depth statistical analyses. In particular, Su et al. (2018) evaluate 18 ImageNet models with respect to their adversarial robustness. Ling et al. (2019) and Dong et al. (2020b) provide platforms to evaluate adversarial attacks. Recently a new line of differentiable robust NAS arose, namely including differentiable network measurements to the one-shot loss target in the DARTS search space (Liu et al., 2019) to increase the robustness (Hosseini et al., 2021; Mok

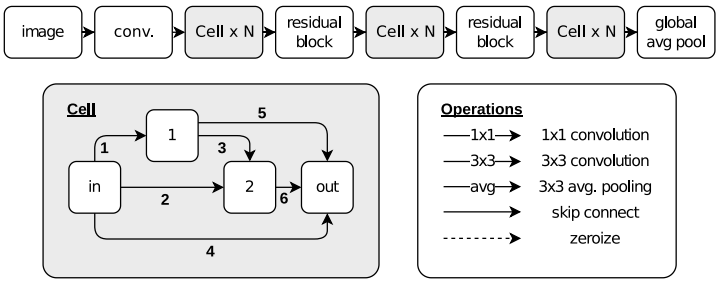


Figure 7.1: **(top)** Macro architecture. Gray highlighted cells differ between architectures, while the other components stay fixed. **(bottom)** Cell structure and the set of possible, predefined operations. (Figure adapted from Dong and Yang (2020))

et al., 2021). Hosseini et al. (2021) define two differentiable metrics to measure the robustness of the architecture, certified lower bound and Jacobian norm bound, and search for architectures by maximizing these metrics, respectively. Mok et al. (2021) propose a search algorithm using the intrinsic robustness of a neural network being represented by the smoothness of the network's input loss landscape, i.e., the Hessian matrix. In this chapter, we will evaluate these architecture metrics for the correlation with the architecture's robustness, i.e., the Jacobian norm as well as the largest eigenvalue of the Hessian matrix. This way we are able to investigate if these differentiable metrics are indeed good measurements for the robustness.

7.3 DATASET GENERATION

7.3.1 *Architectures in NAS-Bench-201*

Recall NAS-Bench-201 is a cell-based architecture search space. Each cell has in total 4 nodes and 6 edges. The nodes in this search space correspond to the architecture's feature maps and the edges represent the architecture's operations, which are chosen from the operation set $O = \{1\times1 \text{ conv.}, 3\times3 \text{ conv.}, 3\times3 \text{ avg. pooling}, \text{skip}, \text{zero}\}$ (see Figure 7.1). This search space contains in total $5^6 = 15\,625$ architectures, from which only 6 466 are unique, since the operations skip and zero can cause isomorphic cells (see Figure C.1), where the latter operation zero stands for dropping the edge. Each architecture is trained on three different image datasets for 200 epochs: CIFAR-10 (Krizhevsky, 2009), CIFAR-100 (Krizhevsky, 2009) and ImageNet16-120 (Chrabaszcz et al., 2017). For our evaluations, we consider all unique architectures in the search space and test splits of the corresponding datasets. Hence, we evaluate $3 \cdot 6\,466 = 19\,398$ pretrained networks in total. In the following, we describe which evaluations we collect.

7.3.2 *Robustness to Adversarial Attacks*

We start by collecting evaluations on different adversarial attacks, namely FGSM, PGD, APGD, and Square Attack. Following, we describe each attack and the collection of their results in more detail.

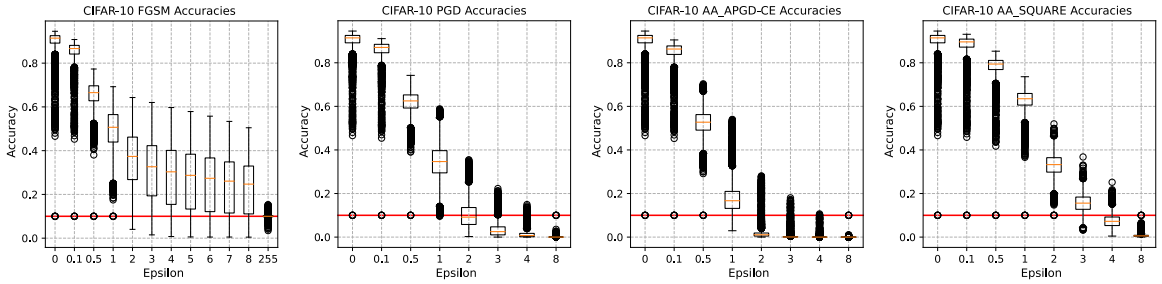


Figure 7.2: Accuracy boxplots over all 6 466 unique architectures in NAS-Bench-201 for different adversarial attacks (FGSM, PGD, APGD, Square) and perturbation magnitude values ϵ, evaluated on CIFAR-10. Red line corresponds to guessing. The large spread indicates towards architectural influence on robust performance.

FGSM

FGSM (Goodfellow et al., 2015) finds adversarial examples via

$$\widetilde{x} = x + \epsilon \operatorname{sign}(\Delta_x J(\theta, x, y)), \tag{7.2}$$

where $\widetilde{x}$ is the adversarial example, x is the input image, y the corresponding label, ϵ the magnitude of the perturbation, and θ the network parameters. $J(\theta, x, y)$ is the loss function used to train the attacked network. In the case of architectures trained for NAS-Bench-201, this is Cross-Entropy (CE). Since attacks via FGSM can be evaluated fairly efficiently, we evaluate all architectures for $\epsilon \in E_{FGSM} = \{.1, .5, 1, 2, \ldots, 8, 255\}/255$, so for a total of $|E_{FGSM}| = 11$ times for each architecture. We use Foolbox (Rauber et al., 2017) to perform the attacks.

PGD

While FGSM perturbs the image in a single step of size ϵ, PGD (Kurakin et al., 2017) iteratively perturbs the image via

$$\widetilde{x}_{n+1} = \operatorname{clip}_{\epsilon,x}(\widetilde{x}_n - \alpha \operatorname{sign}(\Delta_x J(\theta, \widetilde{x}_n, \widetilde{y}))), \widetilde{x}_0 = x, \tag{7.3}$$

where $\widetilde{y}$ is the least likely predicted class of the network, and $\operatorname{clip}_{\epsilon,x}(\cdot)$ is a function clipping the range to $[x - \epsilon, x + \epsilon]$. Due to its iterative nature, PGD is more efficient in finding adversarial examples, but requires more computation time. Therefore, we find it sufficient to evaluate PGD for $\epsilon \in E_{PGD} = \{.1, .5, 1, 2, 3, 4, 8\}/255$, so for a total of $|E_{PGD}| = 7$ times for each architecture. As for FGSM, we use Foolbox (Rauber et al., 2017) to perform the attacks using their L_∞ PGD implementation (see Section 7.1) and keep the default settings, which are $\alpha = 0.01/0.3$ for 40 attack iterations.

APGD

AutoAttack (Croce and Hein, 2020) offers an adaptive version of PGD that reduces its step size over time without the need for hyperparameters. We perform this attack using the L_∞ implementation provided by Croce and Hein (2020) on CE and choose $E_{APGD} = E_{PGD}$. We kept the default number of attack iterations that is 100.

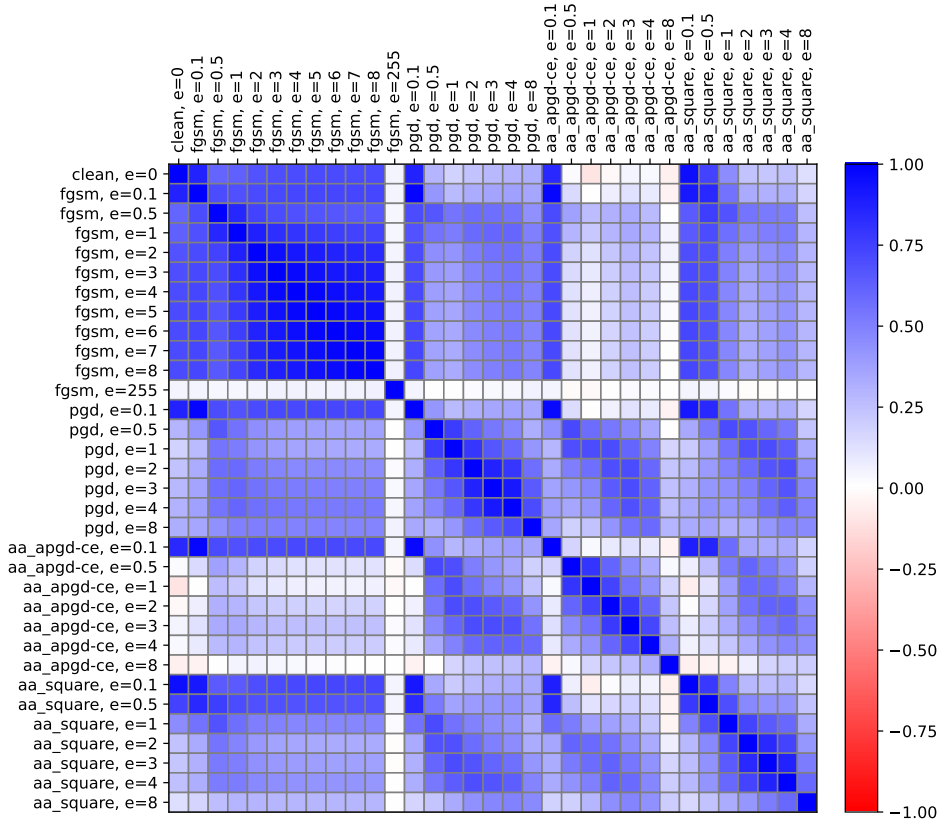


Figure 7.3: Kendall rank correlation coefficient between clean accuracies and robust accuracies on different attacks and magnitude values ϵ on CIFAR-10 for all unique architectures in NAS-Bench-201. There seem to be architectural distinctions for susceptibility to different attacks.

Square Attack

In contrast to the before-mentioned attacks, Square Attack is a black-box attack that has no access to the networks' gradients. It solves the following optimization problem using random search:

$$\begin{aligned} &\min_{\widetilde{x}} \{f_{\theta,y}(\widetilde{x}) - \max_{c \neq y} f_{\theta,c}(\widetilde{x})\} \\ &\text{s.t. } \|\widetilde{x} - x\|_p \leq \epsilon, \end{aligned} \tag{7.4}$$

where $f_{\theta,c}(\cdot)$ are the network predictions for class c given an image. We perform this attack using the L_∞ implementation provided by Croce and Hein (2020) and choose $E_{Square} = E_{PGD}$. We kept the default number of search iterations at 5 000.

We collect for all mentioned adversarial attacks (a) accuracy, (b) average prediction confidences, and (c) confusion matrices for each network and ϵ combination.

Summary

Figure 7.2 shows aggregated evaluation results on the before-mentioned attacks on CIFAR-10 w.r.t. accuracy. Growing gaps between mean and max accuracies indicate that the architecture has

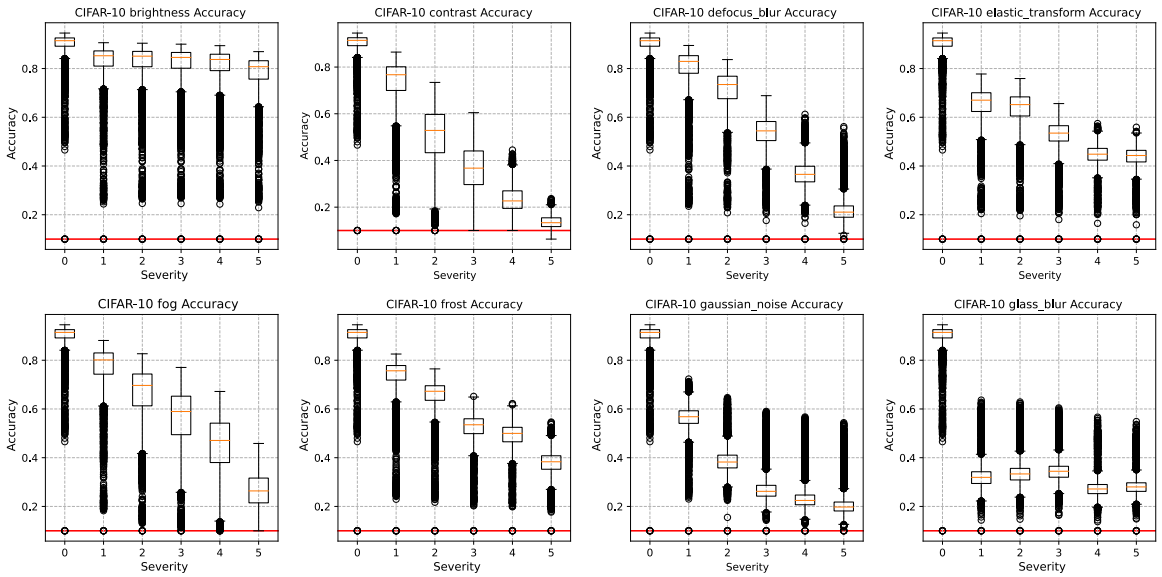


Figure 7.4: Accuracy boxplots over all unique architectures in NAS-Bench-201 for different corruption types at different severity levels, evaluated on CIFAR-10-C. Red line corresponds to guessing. All corruptions can be found in Figure C.14. The large spread indicates towards architectural influence on robust performance.

an impact on robust performances. Figure 7.3 depicts the correlation of ranking all architectures based on different attack scenarios. While there is larger correlation within the same adversarial attack and different values of ϵ, there seem to be architectural distinctions for susceptibility to different attacks.

7.3.3 *Robustness to Common Corruptions*

To evaluate all unique NAS-Bench-201 architectures on common corruptions, we evaluate them on the benchmark data provided by Hendrycks and Dietterich (2019). Two datasets are available: CIFAR10-C, which is a corrupted version of CIFAR-10 and CIFAR-100-C, which is a corrupted version of CIFAR-100. Both datasets are perturbed with a total of 15 corruptions at 5 severity levels (see Figure C.11 for an example). The training procedure of NAS-Bench-201 only augments the training data with random flipping and random cropping. Hence, no influence should be expected of the training augmentation pipeline on the performance of the networks to those corruptions. We evaluate each of the $15 \cdot 5 = 75$ datasets individually for each network and also collect (a) accuracy, (b) average prediction confidences, and (c) confusion matrices.

Summary

Figure 7.4 depicts the mean accuracies for different corruptions at increasing severity levels. Similar to Figure 7.2, a growing gap between mean and max accuracies for most corruptions can be observed, indicating architectural influences on robustness to common corruptions. Figure 7.5 depicts the ranking correlation for all architectures between clean and corrupted accuracies. Ranking architectures based on accuracy on different kinds of corruption is mostly uncorrelated.

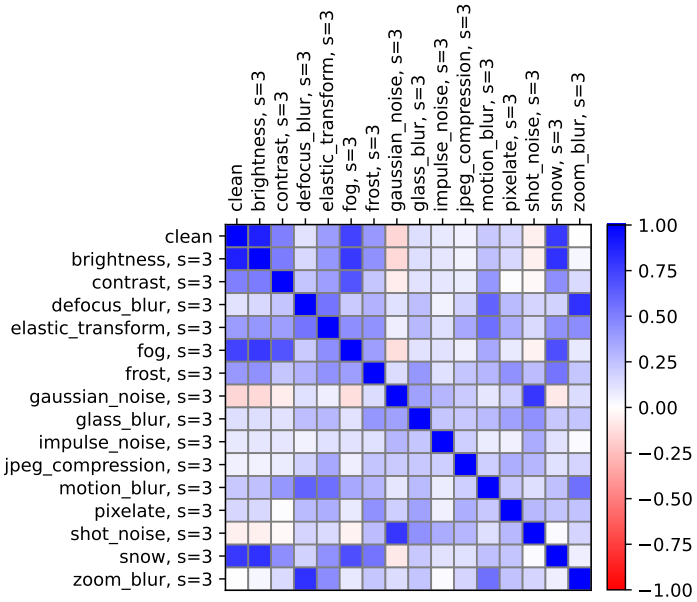


Figure 7.5: Kendall rank correlation coefficient between clean accuracies and accuracies on different corruptions at severity level 3 on CIFAR-10-C for all unique architectures in NAS-Bench-201. The mostly uncorrelated ranking indicates towards high diversity of sensitivity to different kinds of corruption based on architectural design.

This indicates a high diversity of sensitivity to different kinds of corruption based on architectural design.

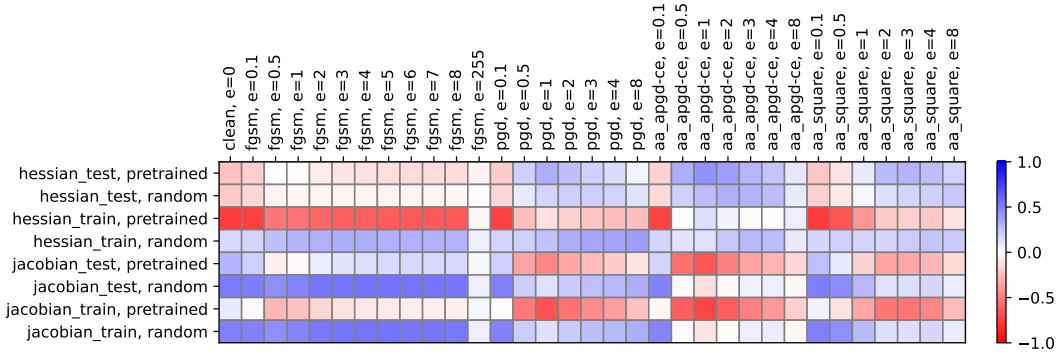
7.4 USE CASES

7.4.1 *Training-Free Measurements for Robustness*

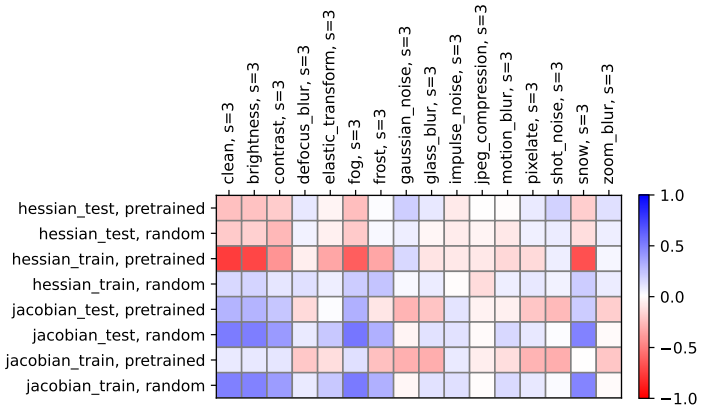
Recently, a new research focus in differentiable NAS shifted towards finding not only high-scoring architectures but also finding adversarial robust architectures against several adversarial attacks (Hosseini et al., 2021; Mok et al., 2021) using training characteristics of neural networks. On the one hand, Hosseini et al. (2021) use Jacobian-based differentiable metrics to measure robustness. On the other hand, Mok et al. (2021) improve the search for robust architectures by including the smoothness of the loss landscape of a neural network. In this section, we evaluate these training-free gradient-based measurements with our dataset. Recall training-free measurements are performance estimation techniques, assigning scores to each architecture based on fast computations, which are supposedly correlated with the final accuracy of the architecture (Mellor et al., 2021).

Background: Jacobian

To improve the robustness of neural architectures, Hoffman et al. (2019) introduce an efficient Jacobian regularization method with the goal to minimize the network's output change in case of perturbed input data, by minimizing the Frobenius norm of the network's Jacobian matrix, $\mathcal{J}$. The



(a) Adversarial attacks.



(b) Common corruptions.

Figure 7.6: Kendall rank correlation coefficient between Jacobian- and Hessian-based robustness measurements computed on all unique NAS-Bench-201 architectures to corresponding rankings given by **(top)** different adversarial attacks and **(bottom)** different common corruptions. Measurements and accuracies are computed on CIFAR-10 / CIFAR-10-C. Measurements are computed on randomly initialized and pretrained networks contained in NAS-Bench-201. Jacobian-based and Hessian-based measurements correlate well for smaller ϵ values, but not for larger ϵ values.

Frobenius norm is defined as $\|\mathcal{J}(x)\|_F = \sqrt{\sum_{\theta,c} |\mathcal{J}_{\theta,c}(x)|^2}$. Let $f_\theta : \mathbb{R}^D \to \mathbb{R}^C$ be a neural network with weights denoted by θ and let $x \in \mathbb{R}^D$ be the input data and $z \in \mathbb{R}^C$ be the output score. Let $\widetilde{x} = x + \varepsilon$ be a perturbed input, with $\varepsilon \in \mathbb{R}^D$ being a perturbation vector. The c-th component of the output of the neural network shifts then to $f_{\theta,c}(x+\varepsilon) - f_{\theta,c}(x)$. The input-output Jacobian matrix can be used as a measurement for the network's stability against input perturbations (Hoffman et al., 2019):

$$f_{\theta,c}(x+\varepsilon) - f_{\theta,c}(x) \approx \sum_{d=1}^{D} \varepsilon_d \cdot \frac{\partial f_{\theta,c}}{\partial x_d}(x) = \sum_{d=1}^{D} \mathcal{J}_{\theta,c;d}(x) \cdot \varepsilon_d, \tag{7.5}$$

according to Taylor-expansion. From Equation (7.5), we can directly see that the larger the Jacobian components, the larger is the output change and thus the more unstable is the neural network against perturbed input data. In order to increase the stability of the network, Hoffman et al. (2019) propose to decrease the Jacobian components by minimizing the square of the Frobenius

norm of the Jacobian. Following Hosseini et al. (2021), we use the efficient algorithm presented in Hoffman et al. (2019) to compute the Frobenius norm based on random projection for each neural network in the NAS-Bench-201 benchmark.

Benchmarking Results: Jacobian

The smaller the Frobenius norm of the Jacobian of a network, the more robust the network is supposed to be. Our dataset allows for a direct evaluation of this statement on all 6 466 unique architectures. We use 10 mini-batches of size 256 of the training as well as test dataset for both randomly initialized and pretrained networks and compute the mean Frobenius norm. The results in terms of ranking correlation to adversarial robustness is shown in Figure 7.6 (top), and in terms of ranking correlation to robustness towards common corruptions in Figure 7.6 (bottom). We can observe that the Jacobian-based measurement correlates well with rankings after attacks by FGSM and smaller ϵ values for other attacks. However, this is not true anymore when ϵ increases, especially in the case of APGD.

Background: Hessian

Zhao et al. (2020) investigate the loss landscape of a regular neural network and robust neural network against adversarial attacks. Let $\mathcal{L}(f_\theta(x))$ denote the standard classification loss of a neural network f_θ for clean input data $x \in \mathbb{R}^D$ and $\mathcal{L}(f_\theta(x+\varepsilon))$ be the adversarial loss with perturbed input data $x+\varepsilon, \varepsilon \in \mathbb{R}^D$. Zhao et al. (2020) provide theoretical justification that the latter adversarial loss is highly correlated with the largest eigenvalue of the input Hessian matrix $H(x)$ of the clean input data x, denoted by $\lambda_{\max}$. Therefore the eigenspectrum of the Hessian matrix of the regular network can be used for quantifying the robustness: large Hessian spectrum implies a sharp minimum resulting in a more vulnerable neural network against adversarial attacks. Whereas in the case of a neural network with small Hessian spectrum, implying a flat minimum, more perturbation on the input is needed to leave the minimum. We make use of Chatzimichailidis et al. (2019) to compute the largest eigenvalue $\lambda_{\max}$ for each neural network in the NAS-Bench-201 benchmark.

Benchmarking Results: Hessian

For this measurement, we calculate the largest eigenvalues of all unique architectures using the Hessian approximation in Chatzimichailidis et al. (2019). We use 10 mini-batches of size 256 of the training as well as test dataset for both randomly initialized and pretrained networks and compute the mean largest eigenvalue. These results are also shown in Figure 7.6. We can observe that the Hessian-based measurement, especially using pretrained weights, correlates well with the rankings after attacks by FGSM and smaller ϵ values for other attacks.

7.4.2 *NAS on Robustness*

In this section, we perform different state-of-the-art NAS algorithms on the clean accuracy and the FGSM ($\epsilon = 1$) robust accuracy in the NAS-Bench-201 search space, and evaluate the best found architectures on all provided introduced adversarial attacks. We apply random search (Li and Talwalkar, 2019), local search (White et al., 2021b), regularized evolution (Real et al., 2019) and

	Method	Test Accuracy ($\epsilon = 1$) ↑ Clean	FGSM	PGD	APGD	Squares	Clean ↑
		CIFAR-10					CF-10-C
	Optimum	94.68	69.24	58.85	54.02	73.61	58.55
Clean	BANANAS (White et al., 2021a)	94.21	**64.25**	41.10	18.62	68.69	55.52
Clean	Local Search (White et al., 2021b)	**94.65**	63.95	**41.17**	**18.74**	**69.59**	**56.90**
Clean	Random Search (Li and Talwalkar, 2019)	94.22	63.38	40.09	17.84	68.40	55.60
Clean	Regularized Evolution (Real et al., 2019)	94.53	63.30	40.23	18.11	68.92	56.21
FGSM	BANANAS (White et al., 2021a)	93.52	66.35	45.59	20.72	68.01	54.88
FGSM	Local Search (White et al., 2021b)	**93.86**	**69.10**	**48.27**	**23.18**	**69.47**	**56.57**
FGSM	Random Search (Li and Talwalkar, 2019)	93.57	67.25	46.15	20.93	68.44	55.10
FGSM	Regularized Evolution (Real et al., 2019)	93.77	68.82	47.99	22.59	69.20	56.11

Table 7.1: Neural Architecture Search on the clean test accuracy and the FGSM ($\epsilon = 1$) robust test accuracy for different state of the art methods on CIFAR-10 in the NAS-Bench-201 search space (mean over 100 runs). Results are the mean accuracies of the best architectures found on different adversarial attacks and the mean accuracy over all corruptions and severity levels in CIFAR-10-C.

BANANAS (White et al., 2021a) with a maximal query amount of 300. The results are shown in Table 7.1. Although clean accuracy is reduced, the overall robustness to all adversarial attacks improves when the search is performed on FGSM ($\epsilon = 1$) accuracy. Local search achieves the best performance, which indicates that localized changes to an architecture design seem to be able to improve network robustness.

7.4.3 *Analyzing the Effect of Architecture Design on Robustness*

In this section, we first depict the best architectures in NAS-Bench-201, then show the effect of parameter count on robustness and the magnitude of potential gains in robustness in a limited parameter count setting, and lastly show the effect of single changes to the best performing architecture according to clean accuracy.

Best Architectures

Figure 7.7 visualizes the best architectures in the NAS-Bench-201 search space in terms of clean accuracy, mean adversarial accuracy (over all attacks and ϵ values described in Section 7.3.2), and mean common corruption accuracy (over all corruptions and severities) on CIFAR-10 and their respective edit distances. The edit distance is defined by the number of changes, either node or edge, to change the graph to the target graph. In the case of NAS-Bench-201 architectures, an edit distance of 1 means that exactly one operation differs between two architectures. So in order to modify the best performing architecture in terms of clean accuracy (#13714) into the best performing architecture according to mean corruption accuracy (#3456), we need to exchange two (out of six) operations: (i) exchange operation 2 from 3×3 convolution to zero and (ii) exchange operation 5 from 1×1 convolution to 3×3 convolution.

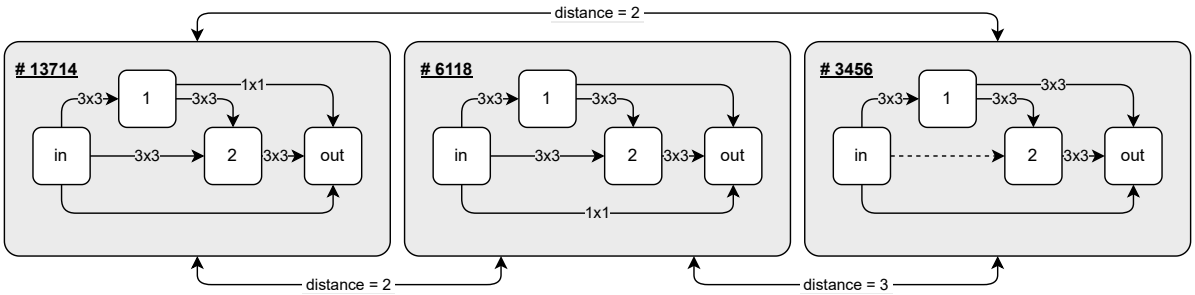


Figure 7.7: Best architectures in NAS-Bench-201 according to (**left**) clean accuracy, (**middle**) mean adversarial accuracy, and (**right**) mean common corruption accuracy on CIFAR-10. See Figure 7.1 for cell connectivity and operations.

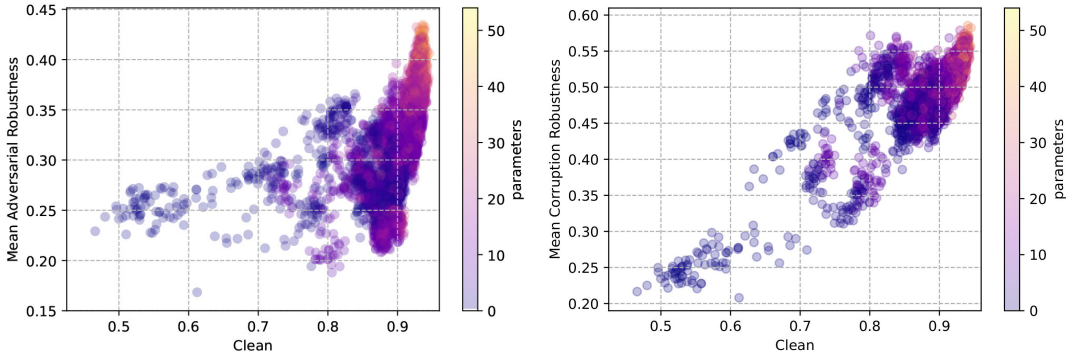


Figure 7.8: (**left**) Mean adversarial robustness accuracies and (**right**) mean corruption robustness accuracies vs. clean accuracies on CIFAR-10 for all unique architectures in NAS-Bench-201. Scatter points are colored based on the number of kernel parameters of a single cell (1 for each 1×1 convolution, 9 for each 3×3 convolution).

Cell Kernel Parameter Count

Figure 7.8 displays the mean adversarial robustness accuracies (left) and the mean corruption robustness accuracies (right) against the clean accuracy, color-coded by the number of cell kernel parameters. We count 1 for each 1×1 convolution and 9 for each 3×3 convolution contained in the cell, hence, their number ranges in $[0, 54]$. Since these are multipliers for the parameter count of the whole network, we coin these *cell kernel parameters*. Overall, we can see that the cell kernel parameter count matters in terms of robustness, hence, that networks with large parameter counts are more robust in general. We can also see that the number of cell kernel parameters are more essential for robustness against common corruptions, where the correlation between clean and corruption accuracy is more linear. Also in terms of adversarial robustness, there seems to be a large magnitude of possible improvements that can be gained by optimizing architecture design.

Limited Cell Parameter Count To further investigate the magnitude of possible improvements via architectural design optimization, we look into the scenario of limited cell parameter count.

In Figure 7.9, we depict all unique architectures in NAS-Bench-201 by their mean adversarial

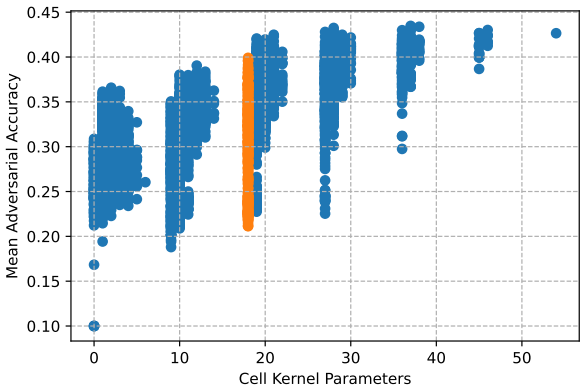


Figure 7.9: Mean robust accuracy on CIFAR-10 by kernel parameters $\in [0, 54]$ for all unique architectures in NAS-Bench-201. Orange scatter points depict all architectures with kernel parameter count 18, hence, architectures with exactly 2 times 3×3 convolutions. Although having exactly the same parameter count, the mean adversarial robustness of these networks ranges in $[0.21, 0.40]$.

robustness accuracy over all attacks and ϵ values as described in Section 7.3.2 and cell kernel parameter count. Networks with parameter count 18 (408 instances in total) are highlighted in orange. As we can see, there is a large range of mean adversarial accuracies $[0.21, 0.4]$ for the parameter count 18 showing the potential of doubling the robustness of a network with *the same parameter count* by carefully crafting its topology.

In Figure 7.10 we show the top-20 performing architectures (color-coded, one operation for each edge) in the mentioned scenario of a parameter count of 18, i.e., exactly 2 times 3×3 convolutions and no 1×1 convolutions, according to mean adversarial accuracy over all attacks and ϵ values (top) and mean corruption accuracy over all corruptions and severities (bottom) on CIFAR-10. It is interesting to see that in both cases, there are (almost) no convolutions on edges 2 and 4, and additionally no dropping (operation zeroize) or skipping (operation skip-connect) of edge 1. In the case of edge 4, it seems that a single convolution layer connecting input and output of the cell increases sensitivity of the network. Hence, most of the top-20 robust architectures stack convolutions (via edge 1, followed by either edge 3 or 5), from which we hypothesize that stacking convolutions operations might improve robustness when designing architectures. At the same time, skipping input to output via edge 4 seems not to affect the robustness negatively, as long as the input feature map is combined with stacked convolutions. We find that optimizing architecture design can have a substantial impact on the robustness of a network. Important to note here is that this potential of doubling the robustness by careful topology crafting is a first observation, which can be made by using our provided dataset. This observation functions as a motivation for how this dataset can be used to analyze robustness in combination with architecture design.

Gains and Losses by Single Changes

The fact that our dataset contains evaluations for all unique architectures in NAS-Bench-201 enables us to analyze the effect of small architectural changes. In Figure 7.11, we depict again all unique architectures by their clean and robust accuracies on CIFAR-10. The red data point in both

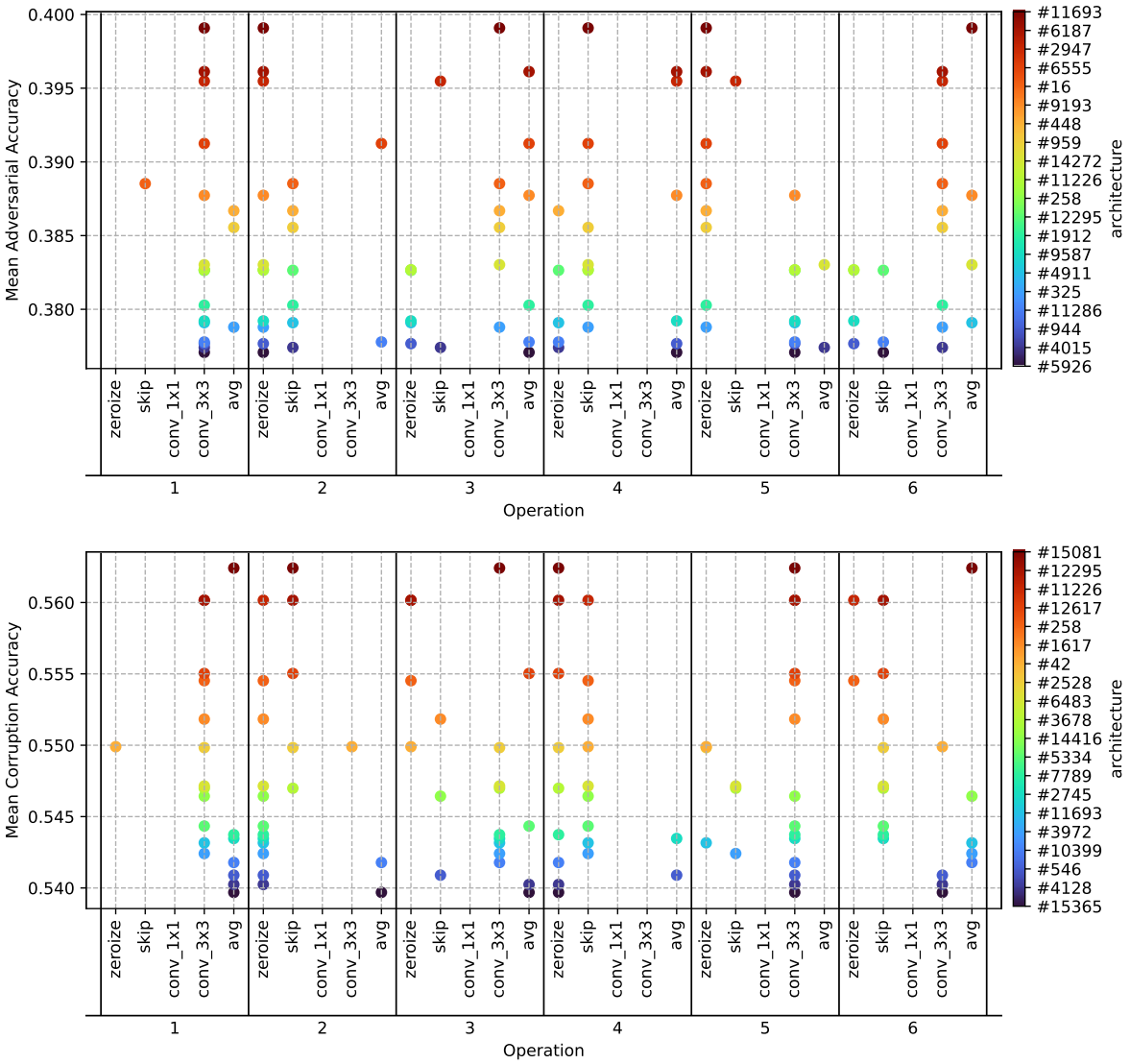


Figure 7.10: Top-20 architectures (out of 408) with cell kernel parameter count 18 (hence, architectures with exactly 2 times 3×3 convolutions and no 1×1 convolutions) according to (**top**) mean adversarial accuracy and (**bottom**) mean corruption accuracy on CIFAR-10. The operation number (1-6) corresponds to the edge in the cell, see Figure 7.1 for cell connectivity and operations. Stacking convolutions seems to be an important part of robust architectural design.

plots shows the best performing architecture in terms of clean accuracy (#13714, see Figure 7.7), while the orange points are its neighboring architectures with edit distance 1. The operation changed for each point is shown in the legend. As we can see in the case of adversarial attacks, we can trade off more robust accuracy for less clean accuracy by changing only one operation. While some changes seem obvious (adding more parameters as with 13 and 14), it is interesting to see that exchanging the 3×3 convolution on edge 3 with average pooling (and hence, reducing the amount of parameters) also improves adversarial robustness. In terms of robustness towards common corruptions, each architectural change leads to worse clean and robust accuracy in this

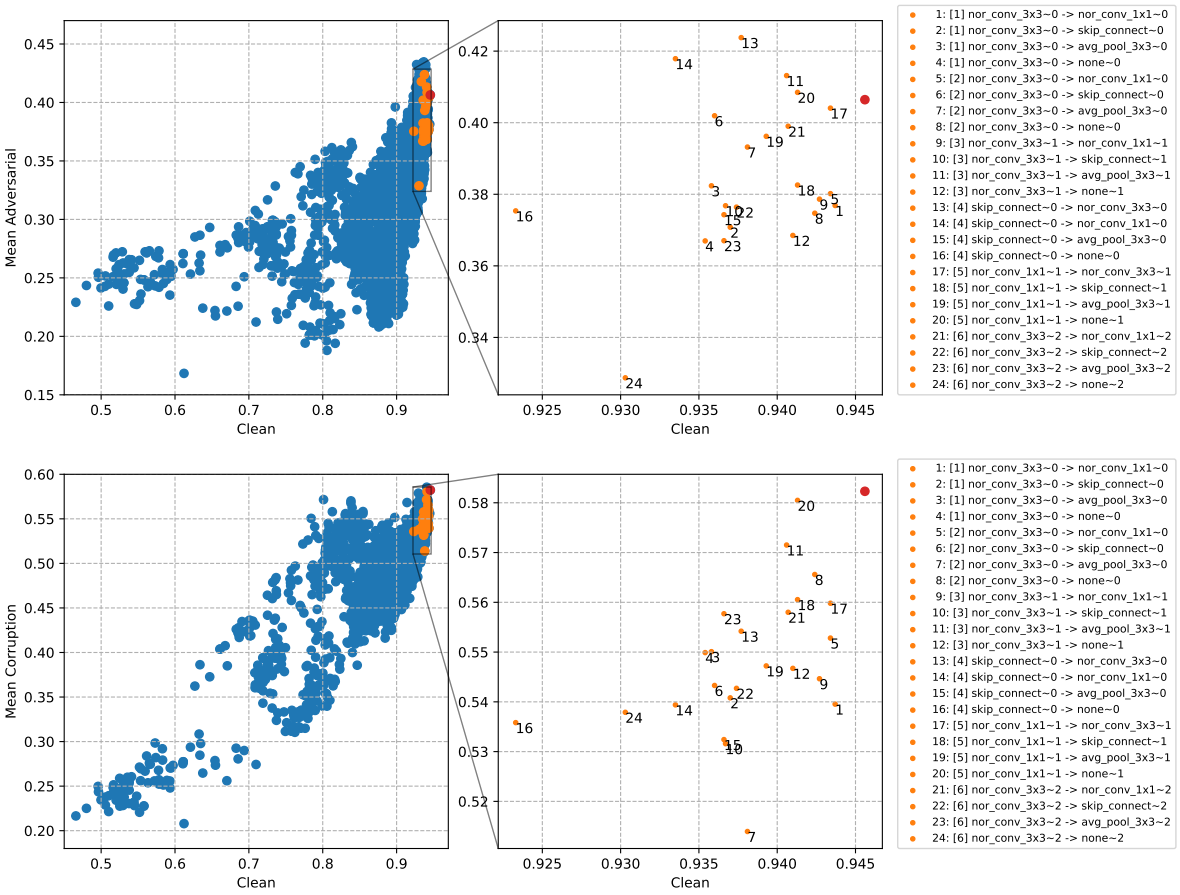


Figure 7.11: (**top**) Scatter plot clean accuracy vs. mean adversarial accuracy on CIFAR-10. (**bottom**) Scatter plot clean accuracy vs. mean common corruption accuracy on CIFAR-10. The red data point shows the best performing architecture according to clean accuracy on CIFAR-10. The orange data points are neighboring architectures, where exactly one operation differs. The change of operation is depicted in the legend. The number in brackets refers to the edge where the operation was changed. See Figure 7.1 for cell connectivity and operations (1-6).

case. Changing more than one operation is necessary to improve common corruption accuracy of this network (as we have seen in Figure 7.7).

7.5 Conclusion

We introduce a dataset for neural architecture design and robustness to provide the research community with more resources for analyzing what constitutes robust networks. We have evaluated *all* 6 466 unique architectures from the commonly used NAS-Bench-201 benchmark against several adversarial attacks and image dataset corruptions. With this full evaluation at hand, we present three use cases for this dataset: First, the correlation between the robustness of the architectures and two differentiable architecture measurements. We show that these measurements are a good first approach for the architecture's robustness, but have to be taken with caution when the perturbation increases. Second, neural architecture search directly on

the robust accuracies, which indeed finds more robust architectures for different adversarial attacks. And last, an initial analysis of architectural design, where we show that it is possible to improve robustness of networks with the same number of parameters by carefully designing their topology.

Part V

Excursus on Graph Clustering

Excursus about Clustering on Graphs

8

In this thesis, we introduced several approaches and contributions to NAS search and estimation strategies, with the goal of more efficient search for not only high-performing (Part II, Part III) but also robust architectures (Part IV), especially with the focus of computer vision tasks and in detail image classification.

Another direction in computer vision tasks involve partitioning (clustering) a set of observations into unique entities. A powerful formulation for such tasks is that of (weighted) correlation clustering (CC). CC is defined on a sparse graph with real valued edge weights, where nodes correspond to observations and weighted edges describe the affinity between pairs of nodes.

For example, in image segmentation (on superpixel graphs), nodes correspond to superpixels and edges indicate adjacency between superpixels. The weight of the edge between a pair of superpixels relates to the probability, as defined by a classifier, that the two superpixels belong to the same ground truth entity. This weight is positive, if the probability is greater than $\frac{1}{2}$ and negative if it is less than $\frac{1}{2}$. The magnitude of the weight is a function of the confidence of the classifier.

The CC cost function sums up the weights of the edges separating connected components (referred to as entities) in a proposed partitioning of the graph. Optimization in CC partitions the graph into entities to minimize the CC cost. CC is appealing, since the optimal number of entities emerges naturally as a function of the edge weights, rather than requiring an additional search over some model order parameter describing the number of clusters (entities) (Yarkony et al., 2012). Optimization in CC is NP-hard for general graphs (Bansal et al., 2004). Previous methods for the optimization of CC problems such as described in Andres et al. (2011) and Andres et al. (2012b) and Nowozin and Jegelka (2009) are based on linear programming with cutting planes. They do not scale easily to large CC problem instances and are not easily parallelizable. The goal of this chapter is to introduce an efficient mechanism for optimization in CC for domains, where massively parallel computation could be employed.

In this chapter, we apply the classic Benders decomposition from operations research (Benders, 1962) to CC for computer vision. Benders decomposition is commonly applied in operations research to solve mixed integer linear programs (MILP) that have a special but common block structure. Benders decomposition partitions the variables in the MILP between a master problem and a set of subproblems. The block structure requires that no row of the constraint matrix of the MILP contains variables from more than one subproblem. Variables explicitly enforced to be integral lie only in the master problem.

Optimization in Benders decomposition is achieved using a cutting plane algorithm. Optimization proceeds with the master problem solving optimization over its variables. The subsequent solution of the subproblems can be done in parallel and provides primal/dual solutions over their variables conditioned on the solution to the master problem. The dual solutions to the subproblems provide constraints to the master problem. Optimization continues until no further constraints are added to the master problem.

Benders decomposition is an exact MILP programming solver, but can be intuitively understood as a coordinate descent procedure, iterating between the master problem and the subproblems. Here, solving the subproblems not only provides a solution for their variables, but

also a lower bound in the form of a hyper-plane over the master problem's variables. This lower bound is tight at the current solution to the master problem.

Benders decomposition is accelerated using the seminal operations research technique of Magnanti-Wong Benders rows (MWR) (Magnanti and Wong, 1981). MWR are generated by solving the Benders subproblems with an alternative (often random) objective under the hard constraint of optimality (possibly within a factor) regarding the original objective of the subproblem.

Our contribution is the use of Benders decomposition with MWR to tackle optimization in CC. This allows for massive parallelization, in contrast to classic approaches to CC such as in Andres et al. (2011).

The remaining chapter is structured as follows: We review the related work in Section 8.1. We provide the standard correlation clustering formulation in Section 8.2 and introduce the formulation of Benders decomposition for correlation clustering in Section 8.3. In Section 8.4 we further derive the Magnanti-Wong Benders rows used in this chapter. The experiments are presented in Section 8.4. Lastly, in Section 8.6 we conclude this chapter.

This chapter presents the paper J. Lukasik et al. (2020b). "A Benders Decomposition Approach to Correlation Clustering". In: *Proc. of the IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*.

8.1 RELATED WORK

Correlation clustering (CC) has been successfully applied to multiple problems in computer vision including image segmentation, multi-object tracking, instance segmentation and multi-person pose estimation. The classical work of Andres et al. (2011) models image segmentation as CC, where nodes correspond to superpixels. Andres et al. (2011) optimize CC using an integer linear programming (ILP) branch-and-cut strategy which precludes parallel execution. Kim et al. (2011) extend CC to include higher-order cost terms over sets of nodes, which they solve using an approach similar to Andres et al. (2011). A parallel optimization scheme for complete, unweighted graphs has been proposed by Pan et al. (2015). This approach relies on random sampling and only provides optimality bounds.

Yarkony et al. (2012) tackle CC in the planar graph structured problems commonly found in computer vision. They introduce a column generation (Gilmore and Gomory, 1961; Barnhart et al., 1996) approach, where the pricing problem corresponds to finding the lowest reduced cost 2-colorable partition of the graph, via a reduction to minimum cost perfect matching (Fisher, 1966; Shih et al., 1990; Kolmogorov, 2009). This approach has been extended to hierarchical image segmentation (Yarkony and Fowlkes, 2015) and to specific cases of non-planar graphs (Yarkony, 2015; Zhang et al., 2014; Andres et al., 2013).

Large CC problem instances such as defined in Keuper et al. (2015b) and Keuper et al. (2015a) and Beier et al. (2016) are usually addressed by primal feasible heuristics (Beier et al., 2014; Beier et al., 2015; Kardoost and Keuper, 2018; Keuper et al., 2015b; Swoboda and Andres, 2017). Such approaches are highly relevant in practice whenever the optimal solution is out of reach, but they do not provide any guarantees on the quality of the solution.

Tang et al. (2015) tackle multi-object tracking using a formulation closely related to CC, where nodes correspond to detections of objects and edges are associated with probabilities of co-association. The work of Insafutdinov et al. (2016) and Pishchulin et al. (2016) build on Tang et al.

(2015) in order to formulate multi-person pose estimation using CC augmented with node labeling.

Our proposed approach is derived from the classical work in operations research on Benders decomposition (Benders, 1962; Birge, 1985; Geoffrion and Graves, 1974). Specifically, we are inspired by the fixed charge formulations of Cordeau et al. (2001), which solves a mixed integer linear program over a set of fixed charge variables (opening links) and a larger set of fractional variables (flows of commodities from facilities to customers in a network) associated with constraints. Benders decomposition reformulates optimization to use only the integer variables and converts the fractional variables into constraints. These constraints are referred to as Benders rows. Optimization is then tackled using a cutting plane approach. Optimization is accelerated by the use of MWR (Magnanti and Wong, 1981), which are more binding than the standard Benders rows.

Benders decomposition has recently been introduced to computer vision (though not for CC), for the purpose of multi-person pose estimation (Wang et al., 2017; Wang et al., 2018; Yarkony and Wang, 2018). In these works, multi-person pose estimation is modeled to admit efficient optimization, using column generation and Benders decomposition jointly. The application of Benders decomposition in our chapter is distinct regarding the problem domain, the underlying integer program and the structure of the Benders subproblems.

8.2 Standard Correlation Clustering Formulation

In this section, we review the standard optimization formulation for CC (Andres et al., 2011), which corresponds to a graph partitioning problem w.r.t. the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. This problem is defined by the following binary edge labeling problem.

Definition 4 (Correlation Clustering). *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $v \in \mathcal{V}$ and undirected edges $(v_i, v_j) \in \mathcal{E}$. A label $x_{v_i v_j} \in \{0, 1\}$ indicates with $x_{v_i v_j} = 1$ that the nodes v_i, v_j are in separate components and is zero otherwise. Given the edge weight $\phi_{v_i v_j} \in \mathbb{R}$, the binary edge labeling problem is to find an edge label $\mathbf{x} = (x_{v_i v_j}) \in \{0, 1\}^{|\mathcal{E}|}$, for which the total weight of the cut edges is minimized:*

$$\min_{\mathbf{x} \in \{0,1\}^{|\mathcal{E}|}} \sum_{(v_i, v_j) \in \mathcal{E}^-} -\phi_{v_i v_j}(1 - x_{v_i v_j}) + \sum_{(v_i, v_j) \in \mathcal{E}^+} \phi_{v_i v_j} x_{v_i v_j} \tag{CC$_1$}$$

$$\text{s.t.} \quad \sum_{(v_i, v_j) \in \mathcal{E}_c^+} x_{v_i v_j} \geq x_{v_i^c v_j^c} \quad \forall c \in C, \tag{8.1}$$

where $\mathcal{E}^-, \mathcal{E}^+$ denote the subsets of $\mathcal{E}$, for which the weight $\phi_{v_i v_j}$ is negative and non-negative, respectively, C is the set of undirected cycles in $\mathcal{E}$ containing exactly one member of $\mathcal{E}^-$, (v_i^c, v_j^c) is the edge in $\mathcal{E}^-$ associated with cycle c and $\mathcal{E}_c^+ \subset \mathcal{E}^+$ is the subset of $\mathcal{E}^+$ associated with cycle c.

Note that the graph $\mathcal{G}$ defined by $\mathcal{E}$ is very sparse for real problems (Yarkony et al., 2012). Also we refer to an edge (v_i, v_j) with $x_{v_i v_j} = 1$ as a *cut* edge.

The objective in Equation (CC$_1$) is to minimize the total weight of the cut edges. The constraints in Equation (8.1) ensure that, within every cycle of $\mathcal{G}$, the number of cut edges can not be exactly one. This enforces the labeling $\mathbf{x}$ to decompose $\mathcal{G}$ such that cut edges are exactly those edges that straddle distinct components. We refer to the constraints in Equation (8.1) as cycle inequalities.

Solving Equation (CC_1) is intractable due to the large number of cycle inequalities. Andres et al. (2011) generate solutions by alternating between solving the ILP over a nascent set of constraints $\widehat{C}$ (initialized empty) and adding new constraints from the set of currently violated cycle inequalities. Generating constraints corresponds to iterating over $(v_i, v_j) \in \mathcal{E}^-$ and identifying the shortest path between the nodes v_i, v_j in the graph with edges $\mathcal{E} \setminus (v_i, v_j)$ and weights equal to $\mathbf{x}$. If the corresponding path has total weight less than $x_{v_i v_j}$, the corresponding constraint is added to $\widehat{C}$. The LP relaxation of Equation (CC_1)-Equation (8.1) can be solved instead of the ILP in each iteration until no violated cycle inequalities exist, after which the ILP must be solved in each iteration.

We should note that earlier work in CC for computer vision did not require that cycle inequalities contain exactly one member of $\mathcal{E}^-$, which is on the right-hand side of Equation (8.1). It is established with Lemma(1) in Yarkony et al. (2014), that the addition of cycle inequalities, that contain edges in $\mathcal{E}^-, \mathcal{E}^+$ on the left-hand side, right-hand side of Equation (8.1), respectively, do not tighten the ILP in Equation (CC_1)-Equation (8.1) or its LP relaxation.

In this section, we reviewed the baseline approach for solving CC in the computer vision community. In the subsequent sections, we rely on the characterization of CC in Equation (CC_1)-Equation (8.1), though not on the specific solver of Andres et al. (2011).

8.3 BENDERS DECOMPOSITION FOR CORRELATION CLUSTERING

In this section, we introduce a novel approach to CC using Benders decomposition (referred to as BDCC).

Our proposed decomposition is defined by a minimal vertex cover on $\mathcal{E}^-$ with members $\mathcal{S} \subset \mathcal{V}$ indexed by v_s. Each $s \in \mathcal{S}$ is associated with a Benders subproblem and v_s is referred to as the root of that Benders subproblem. Edges in $\mathcal{E}^-$ are partitioned arbitrarily between the subproblems, such that each $(v_i, v_j) \in \mathcal{E}^-$ is associated with either the subproblem with root v_i or the subproblem with root v_j. Here, $\mathcal{E}^-_s$ is the subset of $\mathcal{E}^-$ associated with subproblem s. The subproblem with root v_s enforces the cycle inequalities C_s, where C_s is the subset of C containing edges in $\mathcal{E}^-_s$. We use $\mathcal{E}^+_s$ to denote the subset of $\mathcal{E}^+$ adjacent to v_s. In this section, we assume that we are provided with $\mathcal{S}$, which can be produced greedily or using an LP/ILP solver.

Below, we rewrite Equation (CC_1) using an auxiliary function $Q(\phi, s, \mathbf{x})$. Here $Q(\phi, s, \mathbf{x})$ provides the cost to alter $\mathbf{x}$ to satisfy all cycle inequalities in C_s, by increasing/decreasing $x_{v_i v_j}$ for (v_i, v_j) in $\mathcal{E}^+/\mathcal{E}^-_s$, respectively. Below we describe the changes of the master's problem edge labeling $\mathbf{x}$, which is based on the edge labeling of each Benders subproblem $\mathbf{x}^s = (x^s_{v_i v_j}) \in \{0,1\}^{|s|}$, where $|s|$ is the number of edges in the subproblem s:

$$(CC_1) \rightsquigarrow (CC_2): \min_{\mathbf{x} \in \{0,1\}^{|\mathcal{E}|}} \sum_{(v_i,v_j)\in\mathcal{E}^-} -\phi_{v_i v_j}(1 - x_{v_i v_j}) + \sum_{(v_i,v_j)\in\mathcal{E}^+} \phi_{v_i v_j} x_{v_i v_j} + \sum_{s\in\mathcal{S}} Q(\phi, s, \mathbf{x}), \qquad (CC_2)$$

where $Q(\phi, s, \mathbf{x})$ is defined as follows:

$$\begin{aligned} Q(\phi, s, \mathbf{x} = \min_{\mathbf{x}^s \in \{0,1\}^{|s|}} & \sum_{(v_i,v_j)\in\mathcal{E}^-_s} -\phi_{v_i v_j}(1 - x^s_{v_i v_j}) + \sum_{(v_i,v_j)\in\mathcal{E}^+} \phi_{v_i v_j} x^s_{v_i v_j} \\ \text{s.t.} \sum_{(v_i,v_j)\in\mathcal{E}^+_c} & x_{v_i v_j} + x^s_{v_i v_j} \geq x_{v^c_i v^c_j} - (1 - x^s_{v^c_i v^c_j}) \quad \forall c \in C_s. \end{aligned} \qquad (8.2)$$

We now construct a solution $\mathbf{x}^* = \{x^*_{v_i v_j}, (x^{s*}_{v_i v_j})_{s\in\mathcal{S}}\}$ for which Equation (CC_2) is minimized, and all cycle inequalities are satisfied. We start from a given solution $\mathbf{x} = \{x_{v_i v_j}, (x^s_{v_i v_j})_{s\in\mathcal{S}}\}$ and

proceed as follows:

$$x^*_{v_iv_j} \triangleq \min(x_{v_iv_j}, x^s_{v_iv_j}) \quad \forall (v_i, v_j) \in \mathcal{E}^-_s, s \in \mathcal{S} \tag{8.3}$$

$$x^*_{v_iv_j} \triangleq x_{v_iv_j} + \max_{s\in\mathcal{S}} x^s_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^+. \tag{8.4}$$

The right-hand side of Equation (8.4) cannot exceed 1 at optimality because of the constraint in Equation (8.2). Given the solution $x^*_{v_iv_j}$, the optimizing solution to each Benders subproblem s is denoted $x^{s*}_{v_iv_j}$ and is defined as follows:

$$x^{s*}_{v_iv_j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in \mathcal{E}^-_s \\ 0, & \text{otherwise.} \end{cases} \tag{8.5}$$

In Appendix D.1, we show that the cost of $\{x^*_{v_iv_j}, (x^{s*}_{v_iv_j})_{s\in\mathcal{S}}\}$ is no greater than that of $\{x_{v_iv_j}, (x^s_{v_iv_j})_{s\in\mathcal{S}}\}$, with regard to the objective in Equation (CC_2) and that $Q(\phi, s, \mathbf{x}^*) = 0$ holds for all $s \in \mathcal{S}$. It follows that there always exists an optimizing solution $\mathbf{x}$ to Equation (CC_2) such that $Q(\phi, s, \mathbf{x}) = 0$ for all $s \in \mathcal{S}$.

Observe, that there exists an optimal partition $\mathbf{x}^s$ of the nodes of the graph , in Equation (8.2), which is 2-colorable. This is because any partition $\mathbf{x}^s$ can be altered without increasing its cost, by merging connected components that are adjacent to one another, not including the root node v_s. Note, that merging any pair of such components, does not increase the cost, since those components are not separated by negative weight edges in subproblem s and so the result is still a partition.

Given this observation, we rewrite the optimization Equation (CC_2) regarding $Q(\phi, s, \mathbf{x})$, using the node labeling formulation of min-cut, with the notation below.

We indicate with $m_v = 1$ that node $v \in \mathcal{V}$ is not in the component associated with the root of subproblem s and $m_v = 0$ otherwise. To avoid extra notation m_{v_s} is replaced by 0. Let

$$f^s_{v_iv_j} = \begin{cases} 1, & \text{for } (v_i, v_j) \in \mathcal{E}^+, \text{if } (v_i, v_j) \text{ is cut in } \mathbf{x}^s, \text{ but is not cut in } \mathbf{x} \\ 1, & \text{for } (v_i, v_j) \in \mathcal{E}^-_s, \text{if } (v_i, v_j) \text{ is not cut in } \mathbf{x}^s, \text{ but is cut in } \mathbf{x}. \end{cases} \tag{8.6}$$

Thus, the definition for the first/second case implies a penalty of $\phi_{v_iv_j}$/ - $\phi_{v_iv_j}$, which is added to $Q(\phi, s, \mathbf{x})$. Note moreover that $x^s_{v_iv_j} = f^s_{v_iv_j}$ for all $(v_i, v_j) \in \mathcal{E}^+$ and that $x^s_{v_iv_j} = 1 - f^s_{v_iv_j}$ for all $(v_i, v_j) \in \mathcal{E}^-_s$.

Below we write $Q(\phi, s, \mathbf{x})$ as primal/dual LP, with primal constraints associated with dual variables ψ, λ, which are noted in the primal. Given binary $\mathbf{x}$, we only need to enforce that f, m are non-negative to ensure that there exists an optimizing solution for f, m which is binary. This is a consequence of the optimization being totally unimodular, given that $\mathbf{x}$ is binary. Total unimodularity is a known property of the min-cut/max flow LP (Ford and Fulkerson, 1956). The primal subproblem is therefore given by the following:

$$
\begin{aligned}
Q(\phi, s, \mathbf{x}) = \min_{\substack{f^s_{v_iv_j} \geq 0 \\ m_v \geq 0}} & \sum_{(v_i,v_j)\in\mathcal{E}^+} \phi_{v_iv_j} f^s_{v_iv_j} - \sum_{(v_s,v)\in\mathcal{E}^-_s} \phi_{v_sv} f^s_{v_sv} \\
\lambda^-_{v_iv_j} &: m_{v_i} - m_{v_j} \leq x_{v_iv_j} + f^s_{v_iv_j} \quad \forall (v_i, v_j) \in (\mathcal{E}^+ \setminus \mathcal{E}^+_s), \\
\lambda^+_{v_iv_j} &: m_{v_j} - m_{v_i} \leq x_{v_iv_j} + f^s_{v_iv_j} \quad \forall (v_i, v_j) \in (\mathcal{E}^+ \setminus \mathcal{E}^+_s), \\
\psi^-_v &: x_{v_sv} - f^s_{v_sv} \leq m_v \quad \forall (v_s, v) \in \mathcal{E}^-_s, \\
\psi^+_v &: m_v \leq x_{v_sv} + f^s_{v_sv} \quad \forall (v_s, v) \in \mathcal{E}^+_s.
\end{aligned}
\tag{8.7}
$$

This yields to the corresponding dual subproblem:

$$
\begin{aligned}
\max_{\substack{\lambda\geq 0 \\ \psi \geq 0}} & - \sum_{(v_i,v_j)\in(\mathcal{E}^+\setminus\mathcal{E}^+_s)} (\lambda^-_{v_iv_j} + \lambda^+_{v_iv_j}) x_{v_iv_j} + \sum_{(v_s,v)\in\mathcal{E}^-_s} \psi^-_v x_{v_sv} - \sum_{(v_s,v)\in\mathcal{E}^+_s} \psi^+_v x_{v_sv} \\
\text{s.t.} \quad & \psi^+_{v_i} \mathbb{1}_{\mathcal{E}^+_s}(v_s, v_i) - \psi^-_{v_i} \mathbb{1}_{\mathcal{E}^-_s}(v_s, v_i) + \sum_{\substack{v_j \\ (v_i,v_j)\in(\mathcal{E}^+\setminus\mathcal{E}^+_s)}} (\lambda^-_{v_iv_j} - \lambda^+_{v_iv_j}) + \\
& \sum_{\substack{v_j \\ (v_j,v_i)\in(\mathcal{E}^+\setminus\mathcal{E}^+_s)}} (\lambda^+_{v_jv_i} - \lambda^-_{v_jv_i}) \geq 0 \quad \forall v_i \in \mathcal{V} - v_s
\end{aligned}
\tag{8.8}
$$

$$
\begin{aligned}
-\phi_{v_sv} - \psi^-_v &\geq 0 \qquad \forall (v_s, v) \in \mathcal{E}^-_s \\
\phi_{v_sv} - \psi^+_v &\geq 0 \qquad \forall (v_s, v) \in \mathcal{E}^+_s \\
\phi_{v_iv_j} - (\lambda^-_{v_iv_j} + \lambda^+_{v_iv_j}) &\geq 0 \qquad \forall (v_i, v_j) \in (\mathcal{E}^+ \setminus \mathcal{E}^+_s).
\end{aligned}
$$

In Equation (8.8) and subsequently $\mathbb{1}_\Lambda(x)$ denotes the binary indicator function for some set Λ, which returns one if $(x \in \Lambda)$ and zero otherwise. We now consider the constraint that $Q(\phi, s, \mathbf{x}) = 0$. Note that any dual feasible solution for the dual problem Equation (8.8) describes an affine function of $\mathbf{x}$, which is a tight lower bound on $Q(\phi, s, \mathbf{x})$. We compact the terms λ, ψ into ω^z, where $\omega^z_{v_iv_j}$ is associated with the $x_{v_iv_j}$ term.

$$
\omega^z_{v_iv_j} = \begin{cases}
-(\lambda^-_{v_iv_j} + \lambda^+_{v_iv_j}), & \text{if } (v_i, v_j) \in (\mathcal{E}^+ \setminus \mathcal{E}^+_s) \\
-\psi^+_{v_j}, & \text{if } (v_i, v_j) \in \mathcal{E}^+_s \\
\psi^-_{v_j}, & \text{if } (v_i, v_j) \in \mathcal{E}^-_s \\
0, & \text{if } (v_i, v_j) \in (\mathcal{E}^- \setminus \mathcal{E}^-_s).
\end{cases}
$$

We denote the set of all dual feasible solutions across $s \in \mathcal{S}$ as $\mathcal{Z}$, with $z \in \mathcal{Z}$. Observe, that to enforce that $Q(\phi, s, \mathbf{x}) = 0$, it is sufficient to require that $\sum_{(v_i,v_j)\in\mathcal{E}} x_{v_iv_j}\omega^z_{v_iv_j} \leq 0$, for all $z \in \mathcal{Z}$. We formulate CC as optimization using $\mathcal{Z}$ below.

$$
\begin{aligned}
(\text{CC}_2) \rightsquigarrow (\text{CC}_3): \min_{\mathbf{x}\in\{0,1\}^{|\mathcal{E}|}} & \sum_{(v_i,v_j)\in\mathcal{E}^+} \phi_{v_iv_j} x_{v_iv_j} - \sum_{(v_i,v_j)\in\mathcal{E}^-} (1 - x_{v_iv_j}) \phi_{v_iv_j} \\
\text{s.t.} & \sum_{(v_i,v_j)\in\mathcal{E}} x_{v_iv_j}\omega^z_{v_iv_j} \leq 0 \qquad \forall z \in \mathcal{Z}
\end{aligned}
\tag{CC$_3$}
$$

8.3.1 *Cutting Plane Optimization*

Optimization in Equation (CC_3) is intractable since $|\mathcal{Z}|$ equals the number of dual feasible solutions across subproblems, which is infinite. Since we cannot consider the entire set $\mathcal{Z}$, we use a cutting plane approach to construct a set $\widehat{\mathcal{Z}} \subset \mathcal{Z}$, that is sufficient to solve Equation (CC_3) exactly. We initialize $\widehat{\mathcal{Z}}$ as the empty set. We iterate between solving the LP relaxation of Equation (CC_3) over $\widehat{\mathcal{Z}}$ (referred to as the master problem) and generating new Benders rows until no violated constraints exist.

This ensures that no violated cycle inequalities exist but may not ensure that $\mathbf{x}$ is integral. To enforce integrality, we iterate between solving the ILP in Equation (CC_3) over $\widehat{\mathcal{Z}}$ and adding Benders rows to $\widehat{\mathcal{Z}}$. By solving the LP relaxation first, we avoid unnecessary and expensive calls to the ILP solver.

To generate Benders rows given $\mathbf{x}$, we iterate over $\mathcal{S}$ and generate one Benders row using Equation (8.8), if s is associated with a violated cycle inequality, which we determine as follows. Given $s, \mathbf{x}$ we iterate over $(v_i, v_j) \in \mathcal{E}_s^-$. We find the shortest path from v_i to v_j on graph $\mathcal{G}$ with edges $\mathcal{E}$, with weights equal to the vector $\mathbf{x}$. If the length of this path, denoted as $d(v_i, v_j)$, is strictly less than $x_{v_i v_j}$, then we have identified a violated cycle inequality associated with s.

We describe our cutting plane approach in Line 8, with line by line description in Appendix D.2. To accelerate optimization, we add MWR in addition to standard Benders rows, which we describe in the following Section 8.4.

Prior to termination of Line 8, one can produce a feasible integer solution $\mathbf{x}^*$ from any solution $\mathbf{x}$, provided by the master problem, as follows. First, for each $(v_i, v_j) \in \mathcal{E}$, set $x^{**}_{v_i v_j} = 1$, if $x_{v_i v_j} > \frac{1}{2}$ and otherwise set $x^{**}_{v_i v_j} = 0$. Second, for each $(v_i, v_j) \in \mathcal{E}$, set $x^{*}_{v_i v_j} = 1$, if v_i, v_j are in separate connected components of the solution described by $\mathbf{x}^{**}$ and otherwise set $x^{*}_{v_i v_j} = 0$. The cost of the feasible integer solution $\mathbf{x}^*$ provides an upper bound on the cost of the optimal solution. In Appendix D.3, we provide a more involved approach to produce feasible integer solutions.

In this section, we characterized CC using Benders decomposition and provided a cutting plane algorithm to solve the corresponding optimization.

8.4 Magnanti-Wong Benders Rows

We accelerate Benders decomposition (see Section 8.3) using the classic operations research technique of Magnanti-Wong Benders Rows (MWR) (Magnanti and Wong, 1981). The Benders row, given in Equation (8.8), provides a tight bound at $\mathbf{x}^*$, where $\mathbf{x}^*$ is the master problem solution used to generate the Benders row. However, ideally, we want our Benders row to provide good lower bounds for a large set of $\mathbf{x}$ different from $\mathbf{x}^*$, while being tight (or perhaps very active) at $\mathbf{x}^*$. To achieve this, we use a modified version of Equation (8.8), where we replace the objective and add one additional constraint.

We follow the tradition of the operations research literature and use a random negative valued vector (with unit norm) in place of the objective Equation (8.8). This random vector is unique each time a Benders subproblem is solved. We experimented with using as an objective $\frac{-1}{.0001+|\phi_{v_i v_j}|}$, which encourages the cutting of edges with large positive weight, but it works as well as the random negative objective. Here .0001 is a tiny positive number. It prevents the terms in the objective from becoming infinite.

Below, we enforce the new Benders row to be active at $\mathbf{x}^*$, by requiring that the dual cost is

Algorithm 8: Benders Decomposition for Correlation Clustering

Input: (i) $\widehat{\mathcal{Z}} = \{\}$
Input: (ii) done_LP = False
Output: **x**
repeat
  x = Solve Equation (CC_3) over $\widehat{\mathcal{Z}}$ enforcing integrality if and only if done_LP=True
  did_add = False
  for $s \in \mathcal{S}$ **do**
    if $\exists (v_i, v_j) \in \mathcal{E}_s^-$ *s.t.* $d(v_i, v_j) < x_{v_i v_j}$ **then**
      z_1 = Get Benders row via Equation (8.8)
      z_2 = Get MWR via Section 8.4
      $\widehat{\mathcal{Z}} = \widehat{\mathcal{Z}} \cup z_1 \cup z_2$
      did_add = True
    end
  end
  if *did_add=False* **then**
    done_LP = True
  end
until *did_add=False AND* $x_{v_i v_j} \in \{0, 1\}\ \forall (v_i, v_j) \in \mathcal{E}$;

within a tolerance $\tau \in (0, 1)$ of the optimum w.r.t. the objective in Equation (8.8).

$$\tau Q(\phi, s, \mathbf{x}) \leq - \sum_{(v_i, v_j) \in (\mathcal{E}^+ \setminus \mathcal{E}_s^+)} (\lambda^-_{v_i v_j} + \lambda^+_{v_i v_j}) x_{v_i v_j} + \sum_{(v_s, v) \in \mathcal{E}_s^-} \psi_v^- x_{v_s v} - \sum_{(v_s, v) \in \mathcal{E}_s^+} \psi_v^+ x_{v_s v}. \tag{8.9}$$

Here, $\tau = 1$ requires optimality w.r.t. the objective in Equation (8.8), while $\tau = 0$ ignores optimality. In our experiments, we found that $\tau = \frac{1}{2}$ provides strong performance.

8.5 EXPERIMENTS: IMAGE SEGMENTATION

In this section, we demonstrate the value of our algorithm BDCC on CC problem instances for image segmentation on the benchmark Berkeley Segmentation Data Set (BSDS) (Martin et al., 2001). Our experiments demonstrate the following three findings. (1) BDCC solves CC instances for image segmentation; (2) BDCC successfully exploits parallelization; (3) the use of MWR dramatically accelerates optimization.

To benchmark performance, we employ cost terms provided by the OpenGM2 dataset (Andres et al., 2012a) for BSDS. This allows for a direct comparison of our results to the ones from Andres et al. (2011). We use the random unit norm negative valued objective when generating MWR. We use CPLEX to solve all linear and integer linear programming problems considered during the course of optimization. We use a maximum total CPU time of 600 seconds, for each problem instance (regardless of parallelization).

We formulate the selection of $\mathcal{S}$, as a minimum vertex cover problem, where for every edge $(v_i, v_j) \in \mathcal{E}^-$, at least one of v_i, v_j is in $\mathcal{S}$. We solve for the minimum vertex cover exactly as an ILP. Given $\mathcal{S}$, we assign edges in $\mathcal{E}^-$ to a connected selected node in $\mathcal{S}$ arbitrarily. We found

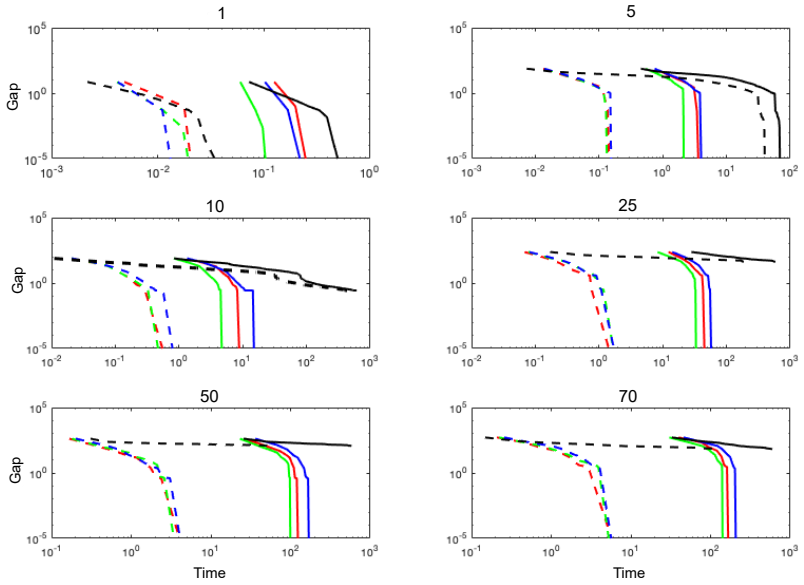


Figure 8.1: We plot the gap between the upper and lower bounds as a function of time for various values of τ on selected problem instances. We use red, green, blue for $\tau = [0.5, 0.99, .01]$ respectively, and black for not using Magnanti-Wong rows. We show both the computation time (in seconds) with and without exploiting parallelization of subproblems with dotted and solid lines, respectively. We use titles to indicate the approximate difficulty of the problem as ranked by input file size of 100 files.

experimentally that solving for the minimum vertex cover consumed negligible CPU time for our dataset. We attribute this fact to the structure of our problem domain, since the minimum vertex cover is an NP-hard problem. For problem instances where solving for the minimum vertex cover exactly is difficult, the minimum vertex cover problem can be solved approximately or greedily.

In Figure 8.1 we demonstrate the effectiveness of BDCC with various τ for different problem difficulties. We observe that the presence of MWR dramatically accelerates optimization. However, the exact value of τ does not affect the speed of optimization dramatically. We show performance with and without relying on parallel processing. Our parallel processing times assume that we have one CPU for each subproblem. For the problem instances in our application the number of subproblems is under one thousand, each of which are very easy to solve. The parallel and non-parallel time comparisons share only the time to solve the master problem. We observe large benefits of parallelization for all settings of τ. However, when MWR are not used, we observe diminished improvement, since the master problem consumes a larger proportion of total CPU time.

In Figure 8.2, we demonstrate the speed-up induced by the use of parallelization. For most problem instances, the total CPU time required when using no MWR was prohibitively large, which is not the case when MWR are employed. Thus, most problem instances are solved without MWR being terminated early.

In Table 8.1, we consider the convergence of the bounds for $\tau = \{0, \frac{1}{2}\}$; ($\tau = 0$ means that no MWR are generated). We consider a set of tolerances on convergence regarding the duality

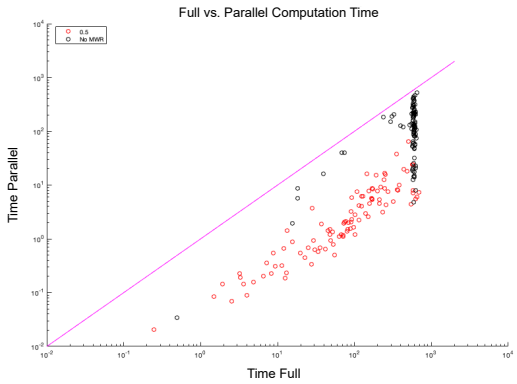


Figure 8.2: We compare the benefits of parallelization and MWR across our data set. We scatter plot the total running time (in seconds) versus the total running time when solving each subproblem is done on its own CPU across problem instances. We use red to indicate $\tau = 0.5$ and black to indicate that MWR are not used. We draw a line with slope=1 in magenta to better enable appreciation of the red and black points. Note, the time spent generating Benders rows, in a given iteration of BDCC when using parallel processing, is the maximum time spent to solve any sub-problem for that iteration.

Tolerance	**MWR** τ	**Parallelization** (par)	**Time in Sec.**			
			10	50	100	300
ϵ=0.1	0.5	0	0.149	0.372	0.585	0.894
	0	0	0.0106	0.0532	0.0745	0.106
	0.5	1	0.266	0.777	0.904	0.968
	0	1	0.0426	0.0745	0.0745	0.138
ϵ=1	0.5	0	0.149	0.394	0.606	0.904
	0	0	0.0106	0.0638	0.0745	0.16
	0.5	1	0.319	0.819	0.947	0.979
	0	1	0.0532	0.0745	0.106	0.17
ϵ=10	0.5	0	0.202	0.426	0.628	0.915
	0	0	0.0532	0.0957	0.128	0.223
	0.5	1	0.447	0.936	0.979	0.989
	0	1	0.0638	0.128	0.181	0.287

Table 8.1: We show the percentage of problems solved that have a duality gap of up to tolerance ϵ, within a certain amount of time (10, 50, 100, 00) seconds, with and without MWR/parallelization. We use par=1 to indicate the use of parallelization and par=0 otherwise. Here $\tau = 0$ means that no MWR are generated.

gap, which is the difference between the anytime solution (upper bound) and the lower bound on the objective. For each such tolerance ϵ, we compute the percentage of instances, for which the duality gap is less than ϵ, after various amounts of time. We observe that the performance of optimization without MWR, but exploiting parallelization performs worse than using MWR, but without paralleliziation. This demonstrates that, across the dataset, MWR are of greater importance than parallelization.

8.6 Conclusions

We present a novel methodology for finding optimal correlation clustering in arbitrary graphs. Our method exploits the Benders decomposition to avoid the enumeration of a large number of cycle inequalities. This offers a new technique in the toolkit of linear programming relaxations, that we expect will find further use in the application of combinatorial optimization to problems in computer vision. The exploitation of results from the domain of operations research may lead to improved variants of BDCC. For example, one can intelligently select the subproblems to solve instead of solving all subproblems in each iteration. This strategy is referred to as partial pricing in the operations research literature. Similarly one can devote a minimum amount of time in each iteration to solve the master problem to enforce integrality on a subset of the variables of the master problem.

Conclusion

9

This thesis introduces several approaches and contributions to Neural Architecture Search. For that, we provide an overview of Neural Architecture Search research in Section 2.1. Since most of our approaches in this thesis rely on Graph Neural Networks, we introduce these in Section 2.2, followed by technical background on Bayesian optimization and variational autoencoder in Section 2.3. In Chapter 3, we introduce an estimation strategy for NAS. Chapter 4 and Chapter 5 present approaches using generative models to improve on the efficiency of NAS methods. In Chapter 6, we analyze the commonly used differentiable architecture search for its ability of a one-shot model. Chapter 7 presents the last chapter for NAS and introduces a dataset for robustness. We furthermore present an excursus to graph decomposition in Chapter 8. In this chapter, we will first summarize this thesis in Section 9.1 in more detail. Secondly, we will provide an outlook on possible future work directions in Section 9.2.

9.1 Summary

Performance Estimation Strategy In Chapter 3, we introduce a surrogate model for performance prediction of neural architectures. This surrogate model is based on Graph Neural Networks, which allows to comprehend and aggregate local node features and graph substructures into one architectural graph representation, resulting in a powerful tool for predicting the performances of architectures, especially for zero-shot prediction.

Generative Architecture Search Chapter 4 and Chapter 5 change the focus towards learned architecture latent space representations, which are then further used for the architecture search. In Chapter 4, we present a variational autoencoder based on Graph Neural Networks to learn a latent space in an unsupervised manner. The proposed "Smooth Variational Graph embedding" model builds a structurally smooth latent space, which allows for competitive search results using Bayesian optimization and surrogate models. The proposed search approach extrapolates from a predefined search space to unseen architectures and can transfer to different datasets with improved accuracy on both tasks.

Inspired by this success of searching for high-performing architectures in learned latent spaces, we take the next step in Chapter 5 and optimize the latent space to directly incorporate the focus of generating high-performing architectures. Therefore, we adapt the decoder from Chapter 4 as a single generative model and couple it with a surrogate model in an end-to-end learning setting. Furthermore, we optimize the latent space, allowing us to reshape the latent space so that high-performing architectures are generated. Therefore, this model does not need classical search approaches as Bayesian optimization and is very efficient. We show this efficiency and ability to find high-performing architectures on several NAS benchmarks and ImageNet. Lastly, this model is extended to allow for joint optimization of architecture performance and hardware properties in a straightforward manner.

One-Shot Architecture Search One-shot methods for architecture search were introduced to improve the search efficiency, with DARTS being the most known. In Chapter 6, we analyze this

widely used differentiable architecture search (DAS) approach in a setting different from image classification, i.e., signal recovery, for its ability as an actual one-shot model. The analysis focuses on domain shifts, hyperparameters sensitivity, the impact of initialization on the search itself, and the problem of rank disorder and poor test generalization. We found that, on the one hand, DAS is able to find well-performing architectures for inverse problems. However, on the other hand, the search outcome depends highly on the hyperparameters, and the rank disorder issue still persists in a well-preconditioned search space. Introducing a different search space even lead to poor test generalization, hampering the search for high-performing architectures.

Robustness in NAS In Chapter 7, we present a dataset for architecture design and robustness to allow for better research of robust network topologies. We evaluate a complete NAS architecture search space against adversarial attacks and corruptions and present several use cases for robustness research. This dataset shows that carefully crafting the architecture design enables the possibility to improve the robustness of architectures substantially and allows for easy access to robustness values to support NAS research in this direction.

Graph Clustering Lastly, we introduce in Chapter 8 an approach for correlation clustering using benders decomposition, known in operations research, which allows for theoretical parallelization.

9.2 OUTLOOK AND FUTURE WORK

Despite the enormous success that NAS has achieved in recent years, especially in the area of image classification, we are still a long way from entirely omitting the human component. In the following, we will address some open problems.

Handcrafted Search Spaces As we have seen in this thesis, especially in Part II, many search spaces exist in the NAS literature. Each is carefully handcrafted to contain well-performing architectures on specific downstream tasks (see Figure 2.5). However, Mehta et al. (2022) already pointed out that popular, supposedly effective search methods do not perform well in all their considered search spaces (28 in total) using the method's default hyperparameters. Therefore, so far, there exists no one best NAS method. The ability of the NAS method to perform well and find high-performing architectures rely heavily on the search space. However, popular existing search spaces are small, like NAS-Bench-101, or constrained in their performance distribution, like DARTS, and limit the possibility of NAS's actual goal: finding novel well-performing architectures.

Search space optimization approaches, as presented in Radosavovic et al. (2020), which starts with a large search space and prunes low-performing areas, are a step towards automating the search space design for NAS.

However, at the same time, it can be beneficial to use the already given handcrafted search spaces, into whose designs a lot of knowledge and computation has already flown, by interpolating between them. A possible approach could be to learn a combined latent space, and a generative model that optimizes the latent space as presented in Chapter 5 or a search method within this latent space using classical methods as Bayesian optimization.

Furthermore, these handcrafted search spaces limit NAS research to domains with several popular and robust search spaces, especially image classification with Convolutional Neural Networks.

Lastly, building such a search space contains many design choices, predominantly introduced by human expert knowledge and focusing on the downstream task. This hinders the possibility of transferring to other domains. A search space should be as generic as possible to allow for transferability. This would eventually lead to larger search spaces, which would, on the one hand, allow for novel architectures but, on the other hand, introduce more enormous search costs.

Hyperparameter Optimization and Architecture Search As we have seen in Chapter 6, the widely used DARTS search approach is quite vulnerable to hyperparameters and the selected predefined search space. In line with related work (Zela et al., 2020a), we have seen that a well-predefined search space is needed to overcome the poor test generalization. At the same time, the original DARTS search space is constrained such that it contains mainly well-performing architectures (see Figure 2.5). However, as discussed, this contradicts the idea of NAS to find new architectures without including much expert knowledge.

Furthermore, the sensitivity to different hyperparameters was also shown in Yang et al. (2020), who tuned hyperparameters and exceeded the performance of NAS found models in the DARTS search space. To overcome the challenge of the two-step procedure of architecture search and hyperparameter optimization, work as Dai et al. (2021) and Zela et al. (2018) seek to combine NAS and hyperparameter optimization. It is important to note that combining these fields is far more challenging than only using either NAS or hyperparameter optimization. The overall search space increases substantially, and each hyperparameter combination can have a different influence on the performance of an architecture, which is, therefore, hard to evaluate.

The ultimate goal is to end up with a fully automated learning approach, which finds a combination of varying architectures along with its optimal hyperparameters to outperform human-found architectures and existing methods.

Robust vs. Task-Oriented Search One-shot models are very popular in NAS, as they allow one to search for a subnetwork from an overparameterized network and also give access to its associated one-shot weights. The first one-shot approach assumed a close correlation between the ranking of the one-shot networks (with their one-shot weight) and the ranking after retraining the architecture from scratch on the downstream task. However, as discussed in this thesis (Section 2.1.4, Chapter 6), this does not hold in general and is also often accompanied by a poor test generalization after retraining. This should result in the careful use of one-shot methods.

In addition, DARTS is based on an approximation of a bilevel optimization problem. Recently, Vicol et al. (2022) investigated bilevel optimization problems in more depth regarding their convergence behavior, especially with the focus of warm-starting the inner level optimization parameter, e.g., the network weights in DARTS, and the question whether to retrain. Popular one-shot models (Pham et al., 2018; Liu et al., 2019; Zela et al., 2020a; Li et al., 2020) rely on retraining the found architecture from scratch to evaluate the found architecture on the downstream tasks correctly due to a possible rank disorder. Single-level optimization, as used in Li et al. (2021b) and Roberts et al. (2021), treats architecture parameters as architecture weights and shows better generalization to different search spaces and downstream tasks without the need for retraining.

However, the goal for NAS using bilevel optimization is to overcome the rank disorder and poor test generalization and find high-performing architectures without retraining.

In general, we can take bilevel optimization one step further in terms of the actual goal we want to pursue. If we want a good-performing architecture on a particular downstream task, retraining of the architecture should not be needed. However, if we want an architecture that is robust, transferable, and generalizable, we want to search for one that does not inherit any performance drop after retraining. Thus, in the long run, using bilevel optimization in architecture search could help find robust and generalizable architectures.

Network biases As we have seen in Chapter 7, designing robust architectures is important for computer vision tasks and should be even further researched in NAS. As already discussed in Section 7.1, the recent research by Geirhos et al. (2019) observed a texture-shape cue-conflict of standard CNNs by analyzing the network's classification behavior. This analysis underlines the importance of understanding and explaining how neural networks behave and how this understanding can further improve the performance of architectures, especially their robustness. Consequently this goal should also be added to the architecture search itself. This way any unwanted bias, as the texture bias, can be reduced, by considering them directly in the search setting, starting with the search space.

Therefore, the next step in NAS is to improve the search for architectures without any unwanted bias and eventually search for more robust and explainable architectures.

So far, we have already taken many steps to search for well-performing architectures on specific downstream tasks in an automated way. However, in the long run, we want NAS to be able to find robust, explainable, and even transferable new architectures with a reduction of the human aspect.

Appendices

Search Space Representation

A

In this section we give more details about the search spaces representations for Part I and Part II.

A.1 NAS-Bench-101

For visualization purposes, we present in Figure A.1 exemplary a DAG from the NAS-Bench-101 (Ying et al., 2019) search space, with its corresponding node attribute matrix and its adjacency matrix. Note, a concatenation of the flatted node attribute matrix and the flatted upper triangular adjacency matrix is the representation our generator model is trained to learn; this holds for all search spaces.

A.2 NAS-Bench-201

Figure A.2 visualizes a DAG in the true variant in the NAS-Bench-201 (Dong and Yang, 2020) space with edge attributes, as well as our adapted representation, where the edge attributes are changed to node attributes. This is similar to the representation in Yan et al. (2020).

A.3 DARTS Search Space

In order to train our generative model to generate valid cells, we additionally randomly sample 500 000 architectures from the DARTS search space. We train our generative model to learn to generate valid cells independently of being a normal or reduction cell. In Figure A.3 we visualize the adapted node attribute matrix and the adapted adjacency matrix to an exemplary DAG in the DARTS search space (Liu et al., 2019). This is similar to the representation in Yan et al. (2020).

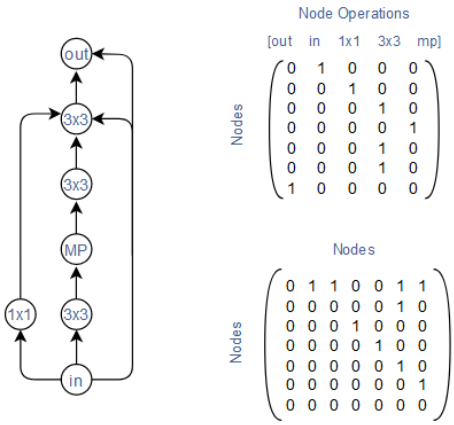


Figure A.1: Exemplary cell representation from the NAS-Bench-101 search space. (**left**) DAG representation of a graph with 7 nodes. (**right**) The top part shows the node attribute matrix to the DAG and the bottom part shows its adjacency matrix.

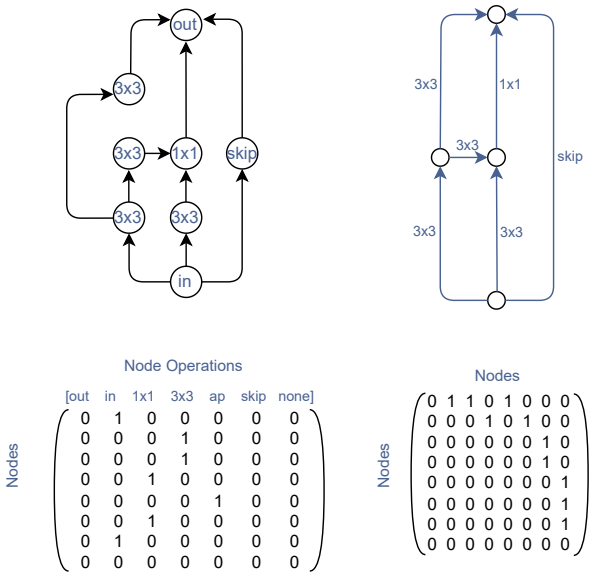


Figure A.2: Exemplary cell representation from the NAS-Bench-201 search space. (**top**) The left part visualizes the DAG representation with node attributes instead of edge attributes. The right part shows the true DAG representation in the NAS-Bench-201 search space. (**bottom**) The left part shows the node attribute matrix to the DAG and the right part shows its adjacency matrix.

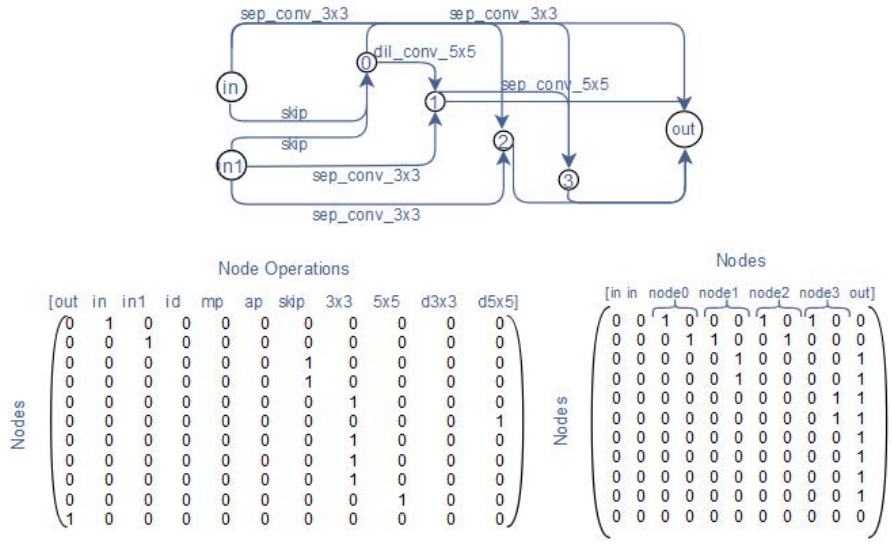


Figure A.3: Exemplary cell representation from the DARTS search space. (**top**) Visualization of the DAG representation in the DARTS search space. (**bottom**) The left part shows the node attribute matrix to the DAG and the right part shows its adjacency matrix.

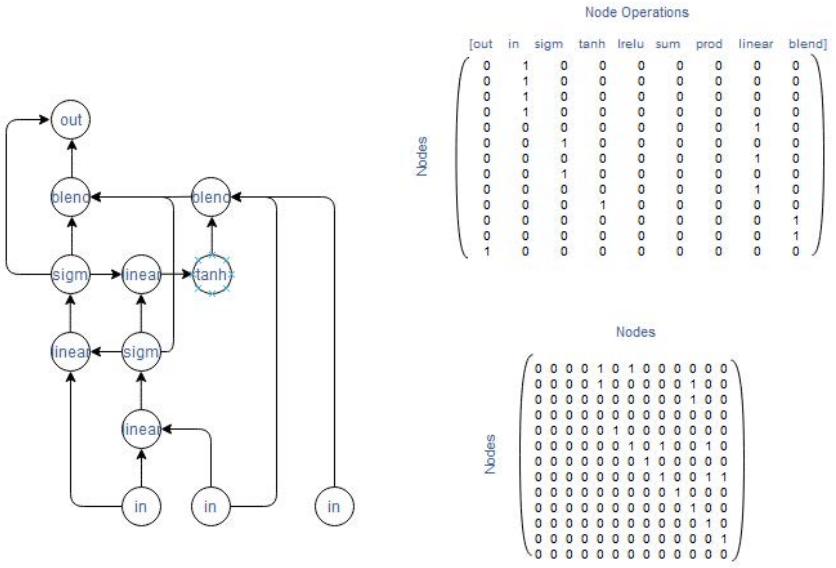


Figure A.4: Exemplary cell representation from the NAS-Bench-NLP search space. (**left**) DAG representation of a graph with 12 nodes. (**right**) The top part shows the node attribute matrix to the DAG and the bottom part shows its adjacency matrix.

A.4 NAS-Bench-NLP

For the experiments on NAS-Bench-NLP (Klyuchnikov et al., 2022) we make use of the surrogate benchmark NAS-Bench-x11 (Yan et al., 2021) and the additional implementation in NAS-Bench-Suite (Mehta et al., 2022). Note, for the NAS-Bench-x11 evaluations, each architecture from the NAS-Bench-NLP search space must be trained for three epochs to use the surrogate model, whereas NAS-Bench-Suite provides the surrogate model for NAS-Bench-NLP without learning curve information, but also accompanying a lower Kendall Tau rank correlation. For fast evaluations we use the latter surrogate for our experiments. In order to use the surrogate benchmark, the architecture representation is the same used in Yan et al. (2021) with the modification that each hidden node is connected to the output node. An exemplary architecture representation is visualized in Figure A.4. A next step is to analyze the 14 332 provided architectures on uniqueness, which leads to 12 107 unique architectures. Furthermore, since Yan et al. (2021) and Mehta et al. (2022) only provide a surrogate model, which only considers architectures with up to 12 nodes, we also restrict our training data to this subset leading to a total of 7 258 architectures.

A.5 Hardware-Aware-NAS-Bench

In our experiments in Section 5.3.4 we consider the latency information on the NAS-Bench-201 search space.

Hyperparameter

In this chapter we provide additional hyperparameter information not mentioned so far in the main part.

B

B.1 Graph Neural Network-based Prediction Model

In this section we give an overview about the hyperparameters for the surrogate model in Chapter 3.

The default hyperparameters are summarized in Table B.1. All our experiments are implemented using PyTorch (Paszke et al., 2019) and PyTorch Geometric (Fey and Lenssen, 2019).

The hyperparameters were tuned with BOHB (Falkner et al., 2018),

Hyperparameter	Default Value
Node Embedding (d_n)	250
Graph Embedding (d_g)	56
GNN layer	2
MLP Regression layer	4
MLP hidden layer	{28,14,7}
Batch Size	128
Optimizer	Adam (Kingma and Ba, 2015)
Learning Rate	0.00005
Epochs	100

Table B.1: Hyperparameters of the GNN surrogate model.

B.2 Variational Autoencoder-based Graph Embeddings

This section provides detailed information about the hyperparameters used in Chapter 4. Also here, we use PyTorch (Paszke et al., 2019) and PyTorch Geometric (Fey and Lenssen, 2019).

B.2.1 *Variational Autoencoder*

The SVGe model is a two-sided GNN-based variational autoencoder. The hyperparameters are described in Table B.2. Note, whenever the loss does not decrease for 10 epochs we multiply the learning rate with 0.1 for both experiments, training the VAE and the surrogate model.

B.2.2 *Surrogate Model*

The overall surrogate is an MLP with ReLU activation functions with hyperparameters listed in Table B.3.

Hyperparameter	Default Value		
	ENAS	NB101	NB201
Node Embedding (one-direction of encoder)	125		
Node Embedding (concatenated)	250		
Graph Embedding	56		
GNN layer	2		
Batch Size	32	128	32
Optimizer	Adam (Kingma and Ba, 2015)		
Learning Rate	0.001		
VAE loss α Equation (4.10)	0.005		
Epochs	300		

Table B.2: Hyperparameters of the autoencoder model.

Hyperparameter	Default Value
MLP Regression layer	4
MLP hidden layer	{28,14,7}
Optimizer	Adam (Kingma and Ba, 2015)
Learning Rate	0.001
Loss proportion β Equation (4.11)	0.1
Epochs	100

Table B.3: Hyperparameters of the latent space surrogate model.

B.3 GENERATIVE NAS WITH LATENT SPACE OPTIMIZATION

In this section we give a detailed overview about the hyperparameters for our generative network from Chapter 5. Also here, we use PyTorch (Paszke et al., 2019) and PyTorch Geometric (Fey and Lenssen, 2019) for all our implementations.

B.3.1 *Generator*

Table B.4 presents all used hyperparameters for the generation training. We train our generator in a ticked manner; after every 5 000 training data, we evaluate our generator for validity ability (see Section 5.3.5). The used pretrained state dict for our search in Section 5.3 is then the one with the highest validation measurement, which is defined by the amount of valid graphs from randomly sampled 10 000 latent vectors $\mathbf{z} \in \mathbb{R}^{32}$ generated to architectures. The training is the same for all different search spaces.

B.3.2 *Surrogate Model*

The overall surrogate is an MLP with ReLU activation functions. Table B.5 and Table B.6 list all hyperparameters for the search experiments in Section 5.3 for the simple performance surrogate model and the multi-objective surrogate model for the additional hardware objective. The hyperparameters for XGB (Chen and Guestrin, 2016) are the same as in Mehta et al. (2022).

For the experiments on the Hardware-Aware Benchmark Section 5.3.4, we implement $g(\cdot)$ equally to the performance predictor $f(\cdot)$, whereas both predictors share weights in our experi-

Hyperparameter	Default Value
Node Embedding	32
Latent Vector	32
MLP Node Embedding layer	2
GNN layer	2
Batch Size	32
Optimizer	Adam (Kingma and Ba, 2015)
Learning Rate	0.0002
Betas	(0.5, 0.999)
Ticks	500
Tick Size	5 000

Table B.4: Hyperparameters of the generator model.

Hyperparameter	Dataset			
	NB101	NB201	NB301	NBNLP
α	0.9			
MLP Layers	4			
MLP Hidden	56	84	176	559
Epochs	15	30	15	30
Optimizer	Adam (Kingma and Ba, 2015)			
LR	0.001			
Betas	(0.5, 0.999)			
weight factor	10 e-3			
batch size	16			
loss	L2			

Table B.5: Hyperparameters for the performance surrogate model $f(\cdot)$.

ments.

B.4 Is Differentiable Architecture Search truly a One-Shot Method?

In this section, we present the hyperparameters used for our experiments in Chapter 6. Table B.8 lists the manually chosen hyperparameters H1 and H2 from Table 6.2. In addition Table B.9 lists all BOHB optimized hyperparameters for the data formations *blur* and *downsampling* as well as the hyperparameters optimized for the final architecture performance and also for the DAS-single method; the search range for the BOHB search is given in the second column. In Table B.10 the BOHB search hyperparameters for the non-sequential search space from Section 6.4.5 are listed. Table B.7 lists all other general hyperparameters used for our experiments in Chapter 6.

B.4.1 *Computational Setup*

All experiments in in Chapter 6 were run on a single `Nvidia GTX 2080ti` graphics card of which two were utilized. The hyperparameter tuning with BOHB was conducted on a single `Nvidia GTX 1080 Ti` graphics card.

Hyperparameter	Hardware-Aware NAS-Bench
α	0.95
λ	0.5
MLP Layers	4
MLP Hidden	82
Epochs	30
Optimizer	Adam (Kingma and Ba, 2015)
LR	0.002
Betas	(0.5, 0.999)
weight factor	10 e-3
penalty term	1000
batch size	16
loss	L2

Table B.6: Hyperparameters for both surrogate models $f(\cdot)$ and $g(\cdot)$ for the multi-objective search in the Hardware-Aware Benchmark.

Hyperparameter	Default Value
Epochs	50
Batch size	128
Noise Level	0.10

Table B.7: General hyperparameters for DAS.

Hyperparameter	H1	H2
Param. learning rate	0.001	0.001
Param. weight decay	$1e-8$	$1e-8$
Param. warm up	False	False
Alpha learning rate	0.001	0.0001
Alpha weight decay	0.001	0.0001
Alpha warm up	True	True
Alpha scheduler	Linear	Linear
Alpha optimizer	Gradient Descent	Gradient Descent

Table B.8: Manually chosen hyperparameters H1 and H2.

B.5 A DATASET FOR NEURAL ARCHITECTURE DESIGN AND ROBUSTNESS

We use the trained architectures from NAS-Bench-201 (Dong and Yang, 2020) with seed 777. The hyperparameter settings for the adversarial attacks from Section 7.3.2 are listed in Table C.1.

Hyperparameter	Search Range	BOHB-one-shot-Blur	BOHB-one-shot-DS	BOHB-Blur	BOHB-DAS-single
Param. learn. rate	$[1e-05, 1]$	0.0014232405	0.0020448382	0.0020882283	0.0014232405
Param. weight decay	$[1e-08, 0.1]$	$8.616e-07$	$5.04e-08$	$4.4e-08$	$8.616e-07$
Param. warm up	[True,False]	False	True	False	False
Alpha learn. rate	$[1e-05, 0.1]$	0.0836808765	0.0100063746	$8.43195e-05$	0.025012337102395577
Alpha weight decay	$[1e-05, 0.1]$	$5.05099e-05$	0.0058022776	0.0127425783	$1.390640076980444e-05$
Alpha warm up	[True, False]	False	True	True	False
Alpha scheduler	[None, Linear]	Linear	Linear	Linear	None
Alpha optimizer	[Adam, Gradient Descent]	Adam	Gradient Descent	Adam	Adam

Table B.9: BOHB optimized hyperparameters for different data formations, objectives and methods.

Hyperparameter	Search Range	BOHB-Non-Seq-one-shot-Blur	BOHB-Non-Seq-Blur
Param. learn. rate	$[1e-05, 1]$	0.0050969066	0.0037014752
Param. weight decay	$[1e-08, 0.1]$	$2.423e-07$	$1.4573e-06$
Param. warm up	[True,False]	False	False
Alpha learn. rate	$[1e-05, 0.1]$	$1.32499e-05$	0.0012395056
Alpha weight decay	$[1e-05, 0.1]$	0.0010171142	0.0002855732
Alpha warm up	[True, False]	False	False
Alpha scheduler	[None, Linear]	None	None
Alpha optimizer	[Adam, Gradient Descent]	Adam	Adam

Table B.10: BOHB optimized hyperparameters for the non-sequential search space for data formation blur and different objectives.

A Dataset for Neural Architecture Design and Robustness

C

In order to ensure reproducibility, we build our dataset on architectures from a common NAS-Benchmark, which we will describe in the following in more detail. In addition, a complete description of the dataset itself is provided in the following. Further, we describe all hyperparameters for reproducing the robustness results. Lastly, a complete description of the dataset itself is in the following. We also provide additional information about mean confidences and confusion matrices for each network in this dataset. Lastly, we provide additional results on other image datasets than CIFAR-10, i.e. CIFAR-100 and ImageNet16-120.

C.1 NAS-Bench-201

We base our evaluations on the NAS-Bench-201 (Dong and Yang, 2020) search space. It is a cell-based architecture search space. Each cell has in total 4 nodes and 6 edges. The nodes in this search space correspond to the architecture's feature maps and the edges represent the architecture's operations, which are chosen from the operation set $O = \{1 \times 1 \text{ conv.}, 3 \times 3 \text{ conv.}, 3 \times 3 \text{ avg. pooling}, \text{skip}, \text{zero}\}$ (see Figure 7.1). This search space contains in total $5^6 = 15\,625$ architectures, from which only 6 466 are unique since the operations skip and zero can cause isomorphic cells (see Figure C.1), where the latter operation zero stands for dropping the edge. Each architecture is trained on three different image datasets for 200 epochs: CIFAR-10 (Krizhevsky, 2009), CIFAR-100 (Krizhevsky, 2009) and ImageNet16-120 (Chrabaszcz et al., 2017). For our evaluations, we consider all unique architectures in the search space and test splits of the corresponding datasets. Hence, we evaluate $3 \cdot 6\,466 = 19\,398$ pretrained networks in total.

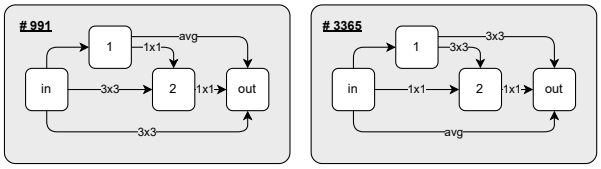


Figure C.1: Example of two isomorphic graphs in NAS-Bench-201. Due to the skip connection from node `in` to node `1`, both computational graphs are equivalent, but their identification in the search space is different. For this dataset, we evaluated all non-isomorphic graphs (`#991` was evaluated and `#3365` was not).

C.2 Dataset Gathering

We collect evaluations for our dataset for different corruptions and adversarial attacks (as discussed in Section 7.3.2 and Section 7.3.3) following Algorithm 9. This process is also depicted in Figure C.2. Hyperparameter settings for adversarial attacks are listed in Table C.1. Due to the heavy load of running all these evaluations, they are performed on several clusters. These clusters are comprised of either (i) compute nodes with Nvidia A100 GPUs, 512 GB RAM, and Intel Xeon IceLake-SP processors, (ii) compute nodes with NVIDIA Quadro RTX 8000 GPUs, 1024 GB RAM, and AMD EPYC 7502P processors, (iii) NVIDIA Tesla A100 GPUs, 2048 GB RAM, Intel Xeon Platinum

8360Y processors, and (iv) NVIDIA Tesla A40 GPUs, 2048 GB RAM, Intel Xeon Platinum 8360Y processors.

Attack	**Hyperparameters**
FGSM	$\epsilon \in \{.1, .5., 1, 2, 3, 4, 5, 6, 7, 8, 255\}/255$
PGD	$\epsilon \in \{.1, .5., 1, 2, 3, 4, 8, 255\}/255$ $\alpha = 0.01/0.3$ 40 attack iterations
APGD	$\epsilon \in \{.1, .5., 1, 2, 3, 4, 8, 255\}/255$ 100 attack iterations
Square	$\epsilon \in \{.1, .5., 1, 2, 3, 4, 8, 255\}/255$ 5 000 search iterations

Table C.1: Hyperparameter settings of adversarial attacks evaluated.

Algorithm 9: Robustness Dataset Gathering.

Input: (i) Architecture space A (NAS-Bench-201).
Input: (ii) Test datasets D (CIFAR-10, CIFAR-100, ImageNet16-120).
Input: (iii) Set of attacks and/or corruptions C.
Input: (iv) Robustness Dataset R.

for $a \in A$ **do**
  ▷ `Load pretrained weights for` a.
  a.load_weights(d)
  for $d \in D$ **do**
    for $c(\cdot,\cdot) \in C$ **do**
      ▷ `Corrupt dataset` d.
      $d_c \leftarrow c(a, d)$
      ▷ `Evaluate architecture` a `with` d_c.
      Accuracy, Confidence, ConfusionMatrix $\leftarrow eval(a, d_c)$
      ▷ `Extend robustness dataset with evaluations.`
      $R[d][c]$["accuracy"]$[a] \leftarrow$ Accuracy
      $R[d][c]$["confidence"]$[a] \leftarrow$ Confidence
      $R[d][c]$["cm"]$[a] \leftarrow$ ConfusionMatrix
    end
  end
end

C.3 DATASET STRUCTURE, DISTRIBUTION, AND LICENSE

Files are provided in `json` format to ensure platform independence and to reduce the dependency on external libraries (e.g., Python has built-in `json`-support).

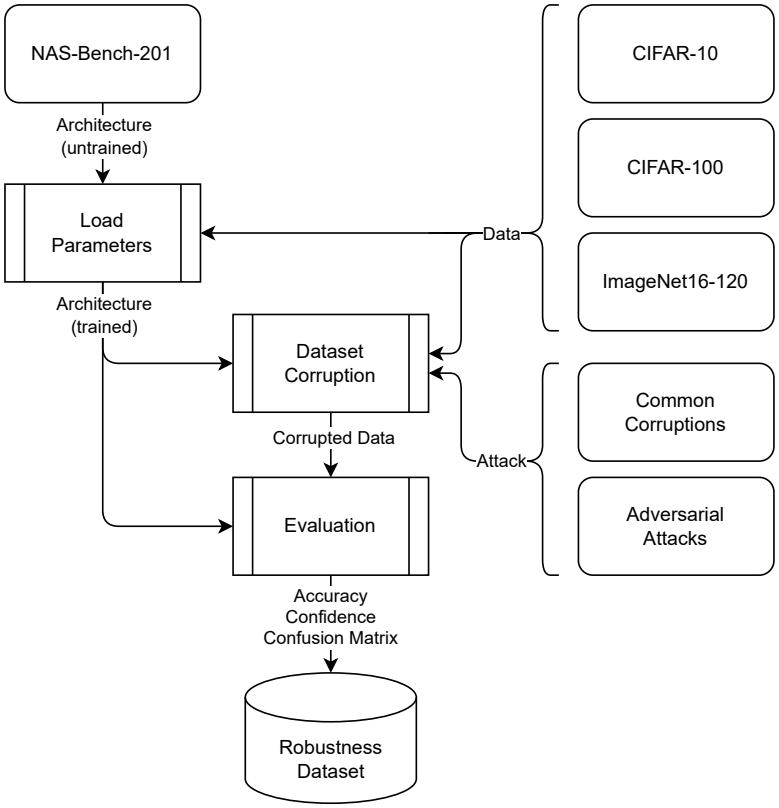


Figure C.2: Diagram showing the gathering process for our robustness dataset. (i) An non-isomorphic architecture contained in NAS-Bench-201 is created and its parameters are loaded from a provided checkpoint, dependent on the dataset evaluated. (ii) Given the evaluation dataset, an attack or corruption, and the trained network, the evaluation dataset is corrupted and (iii) the resulting corrupted data is used to evaluate the network. (iv) The evaluation results are stored in our robustness dataset.

C.4 Structure

The dataset consists of 3 folders, one for each dataset evaluated (`cifar10`, `cifar100`, `ImageNet16-120`). Each folder contains one `json` file for each combination of key and measurement. Keys refer to the sort of attack or corruption used (Table C.2 lists all keys). Measurements refer to the collected evaluation type (`accuracy`, `confidence`, `cm`). Clean and adversarial evaluations are performed on all datasets, while common corruptions are evaluated on `cifar10` and `cifar100`. Additionally, the dataset contains one metadata file (`meta.json`).

Metadata The `meta.json` file contains information about each architecture in NAS-Bench-201. This includes, for each architecture identifier, the corresponding string defining the network design (as per Dong and Yang (2020)) as well as the identifier of the corresponding non-isomorphic architecture from Dong and Yang (2020) that we evaluated. The file also contains all ϵ values that we evaluated for each adversarial attack. An excerpt of this file is shown in Figure C.3.

Clean	Adversarial	Common Corruptions
clean	aa_apgd-ce	brightness
	aa_square	contrast
	fgsm	defocus_blur
	pgd	elastic_transform
		fog
		frost
		gaussian_noise
		glass_blur
		impulse_noise
		jpeg_compression
		motion_blur
		pixelate
		shot_noise
		snow
		zoom_blur

Table C.2: Keys for attacks and corruptions evaluated.


```
{
  "ids": {
    ...,
    "21": {
      "nb201_string": "|nor_conv_1x1~0|+|none~0|none~1|+|nor_conv_1x1~0|nor_conv_3x3~1|none~2|",
      "isomorph": "21"
    },
    ...,
    "1832": {
      "nb201_string": "|nor_conv_1x1~0|+|nor_conv_1x1~0|none~1|+|nor_conv_1x1~0|skip_connect~1|none~2|",
      "isomorph": "309"
    },
    ...
  },
  "epsilons": {
    "aa_apgd_ce": [0.1, 0.5, 1.0, 2.0, 3.0, 4.0, 8.0],
    "aa_square": [0.1, 0.5, 1.0, 2.0, 3.0, 4.0, 8.0],
    "fgsm": [0.1, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 255.0],
    "pgd": [0.1, 0.5, 1.0, 2.0, 3.0, 4.0, 8.0]
  }
}
```


Figure C.3: Excerpt of `meta.json` showing meta information of architectures #21 and #1832, as well as ϵ values for each attack. Architecture #21 is non-isomorphic and points to itself, while architecture #1832 is an isomorphic instance of #309.

Files All files are named according to "{key}_{measurement}.json". Hence, the path to all clean accuracies on cifar10 is "./cifar10/clean_accuracy.json". An excerpt of this file is shown in Figure C.4. Each file contains nested dictionaries stating the dataset, evaluation key and measurement type. For evaluations with multiple measurements, e.g., in the case of adversarial attacks for multiple ϵ values, the results are concatenated into a list. Files and their possible contents are described in Table C.3.

We showed some analysis and possible use-cases on accuracies in the main paper. In the following, we elaborate on and show `confidence` and confusion matrix (`cm`) measurements.

```
{
  "cifar10": {
    "clean": {
      "accuracy": {
        "0": 0.856,
        ...
    }
  }
}
```

```
{
  "cifar10": {
    "pgd": {
      "accuracy": {
        "0": [0.812, 0.582, 0.295, 0.034, 0.002, 0.0, 0.0],
        ...
    }
  }
}
```


Figure C.4: Excerpt of **(left)** `clean_accuracy.json` and **(right)** `pgd_accuracy.json` for dataset `cifar10` for the architecture #0. Numbers are rounded to improve readability.

File	Description
clean_accuracy	one accuracy value for each evaluated network
clean_confidence	one confidence matrix for each evaluated network and collection scheme
clean_cm	one confusion matrix for each evaluated network
{attack}_accuracy	list of accuracies, where each element corresponds to the respective ϵ value
{attack}_confidence	list of confidence matrices, where each element corresponds to the respective ϵ value
{attack}_cm	list of confusion matrices, where each element corresponds to the respective ϵ value
{corruption}_accuracy	list of accuracies, where each element corresponds to the respective corruption severity
{corruption}_confidence	list of confidence matrices, where each element corresponds to the respective corruption severity
{corruption}_cm	list of confusion matrices, where each element corresponds to the respective corruption severity

Table C.3: Files and their possible content.

C.5 Confidence

We collect the mean confidence after softmax for each network over the whole (attacked) test dataset evaluated. We used 3 schemes to collect confidences (see Figure C.6). First, confidences for each class are given by true labels (called `label`). In case of `cifar10`, this results in a 10×10 confidence matrix, for `cifar100` a 100×100 confidence matrix, and `ImageNet16-120` a 120×120 confidence matrix. Second, confidences for each class are given by the class predicted by the network (called `argmax`). This again results in matrices of sizes as mentioned. Third, confidences for correctly classified images as well as confidences for incorrectly classified images (called `prediction`). For all image datasets, this results in a vector with 2 dimensions. Each result is saved as a list (or list of list), see Figure C.5.

Figure C.7 shows a progression of `label` confidence values for class label 0 on `cifar10` from `clean` to `fgsm` with increasing values of ϵ. Figure C.8 shows how `prediction` confidences of

correctly and incorrectly classified images correlate with increasing values of ϵ when attacked with `fgsm`.

```
{
  "cifar10": {
    "clean": {
      "confidence": {
        "0": {
          "label": [[...]],
          "argmax": [[...]],
          "prediction": [...]
        }
    }
  }
}
```

Figure C.5: Excerpt of `clean_confidence.json` for `cifar10`. Numbers are not shown to improve readability.

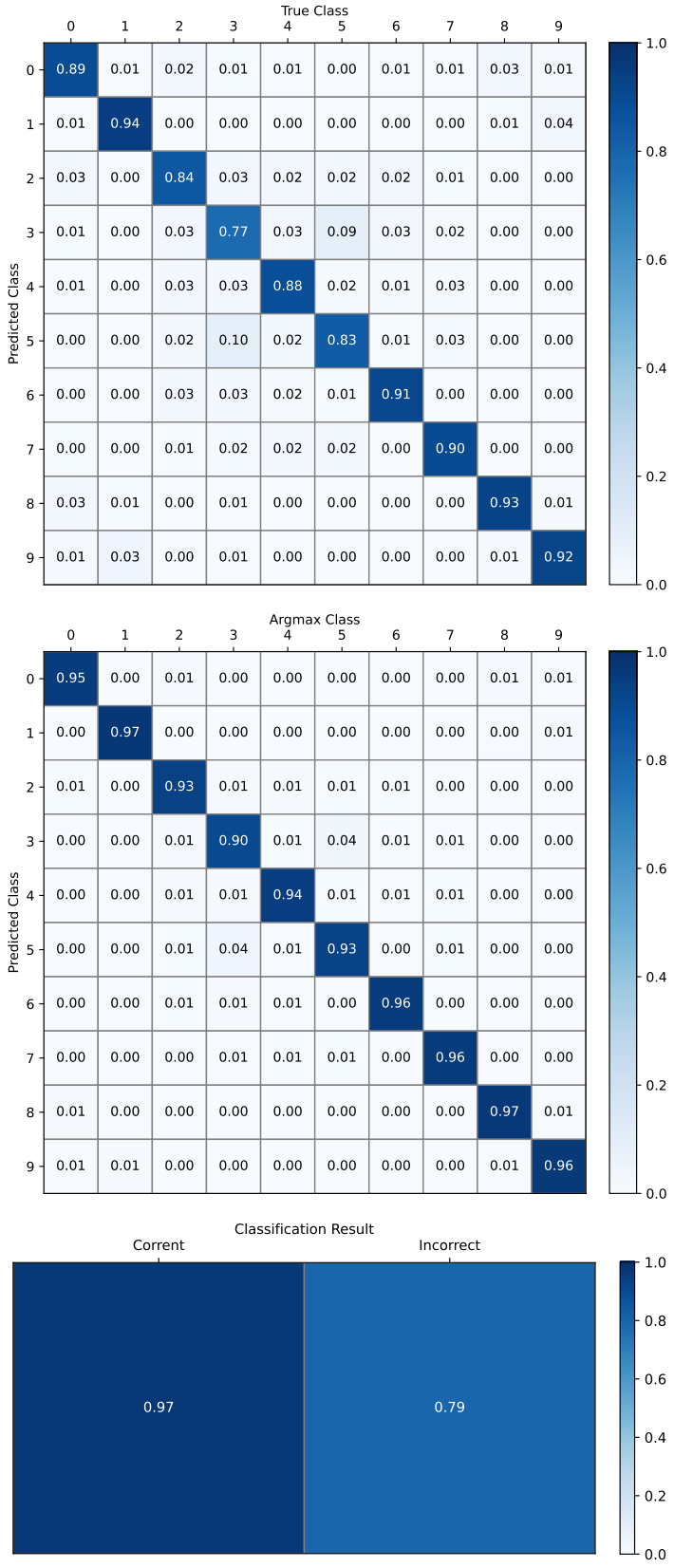


Figure C.6: Mean confidence scores on clean CIFAR-10 images for all non-isomorphic networks in NAS-Bench-201. (**top**: `label`) For each true class label. (**middle**: `argmax`) For each predicted class label. (**bottom**: `prediction`) For correct and incorrect classifications.

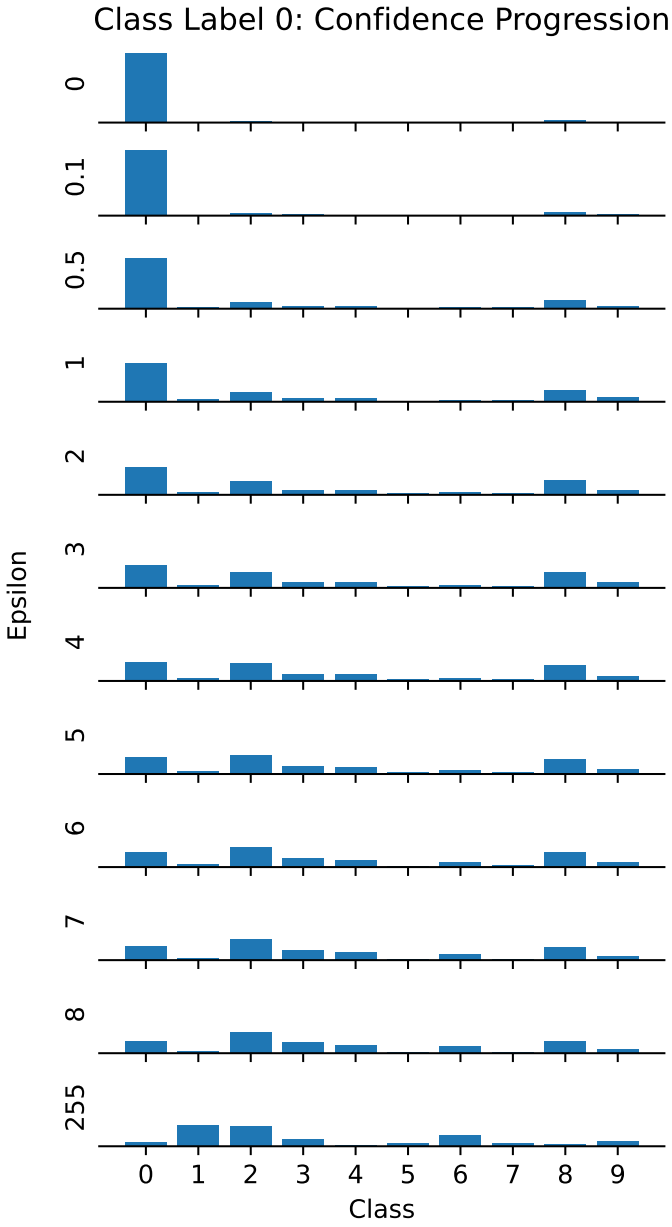


Figure C.7: Mean `label` confidence scores on FGSM-attacked CIFAR-10 images for different ϵ for all non-isomorphic networks in NAS-Bench-201. Only confidence scores for class label 0 are shown. Networks lose prediction confidence for the true label when ϵ increases.

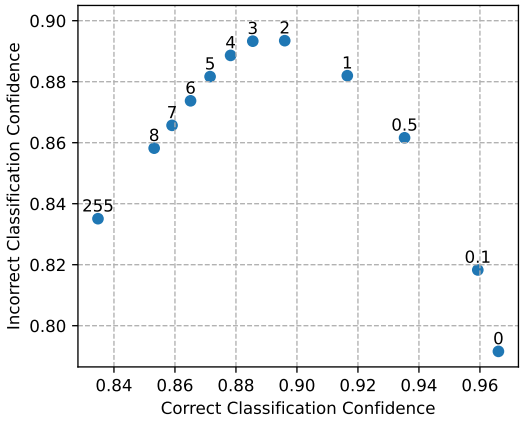


Figure C.8: Mean `prediction` confidence scores on FGSM-attacked CIFAR-10 images for different ϵ (on top of points) for all non-isomorphic networks in NAS-Bench-201. Networks become less confident in their prediction if their prediction is correct when ϵ increases. Networks become more confident in their prediction if their prediction is incorrect, however, only up to a certain ϵ value. When ϵ further increases, confidence drops again.

C.6 Confusion Matrix

For each evaluated network, we collect the confusion matrix (key: `cm`) for the corresponding (attacked) test dataset. The result is a 10×10 matrix in case of `cifar10`, a 100×100 matrix in case of `cifar100`, and a 120×120 matrix in case of `ImageNet16-120`. See Figure C.9 for an example, where we summed up confusion matrices for all networks on `cifar10`.

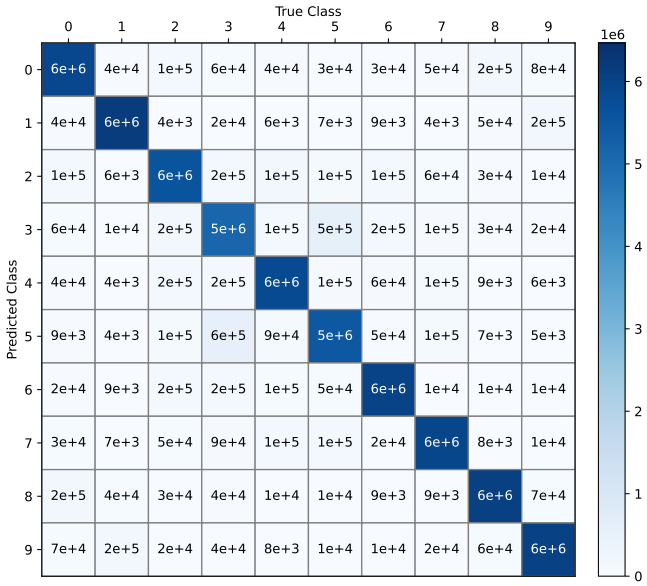


Figure C.9: Aggregated confusion matrices on clean CIFAR-10 images for all non-isomorphic networks in NAS-Bench-201.

C.7 CORRELATIONS BETWEEN IMAGE DATASETS

In Figure C.10 we show the correlation between all clean and adversarial accuracies over all datasets collected. This plot shows a positive correlation between the image datasets for the one-step FGSM attack, whereas for all other multi-step attacks, the correlation becomes close to zero or even negative.

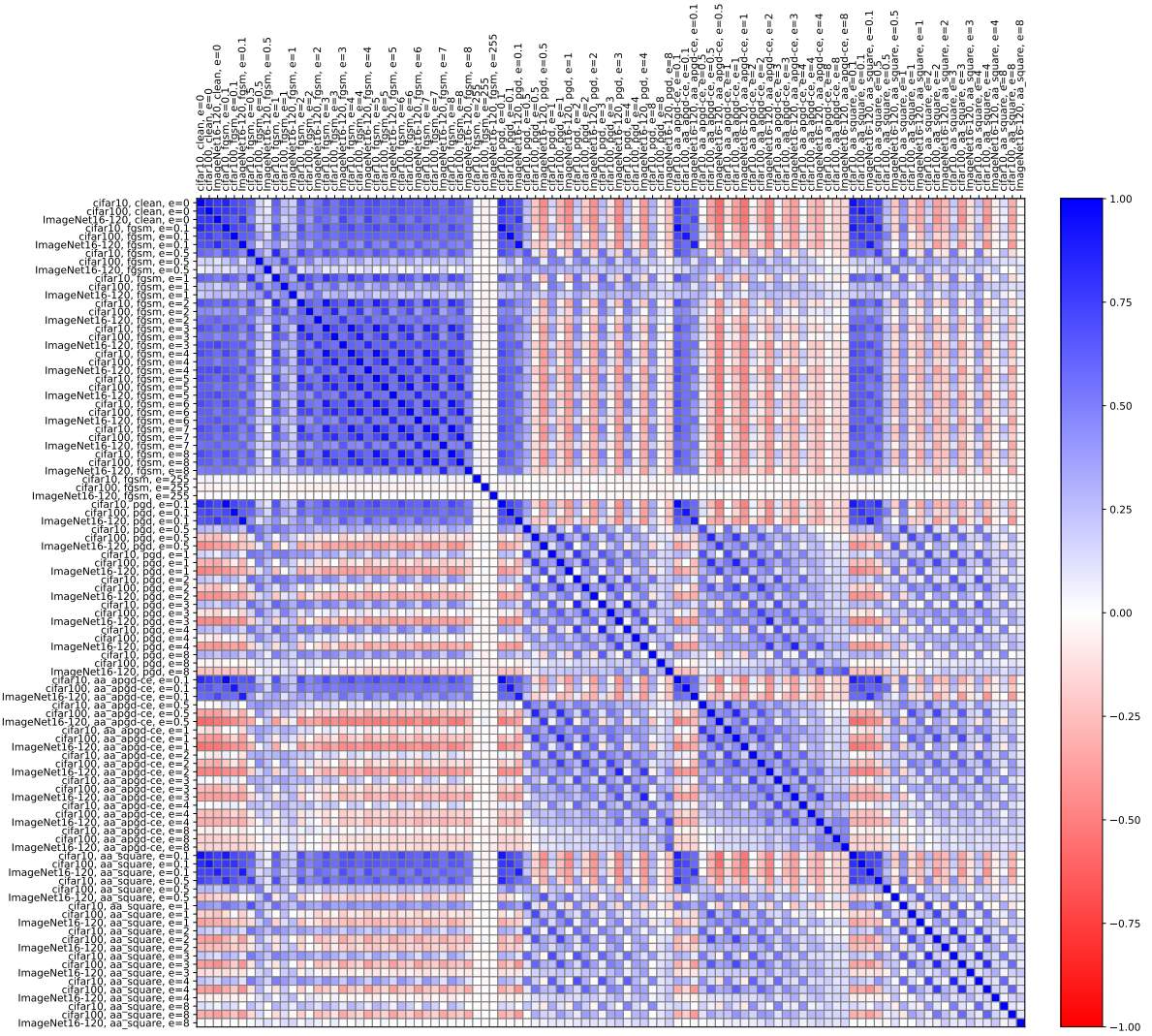


Figure C.10: Kendall rank correlation coefficient between all clean and adversarial accuracies that are evaluated in our dataset.

C.8 Example image of corruptions in CIFAR-10-C

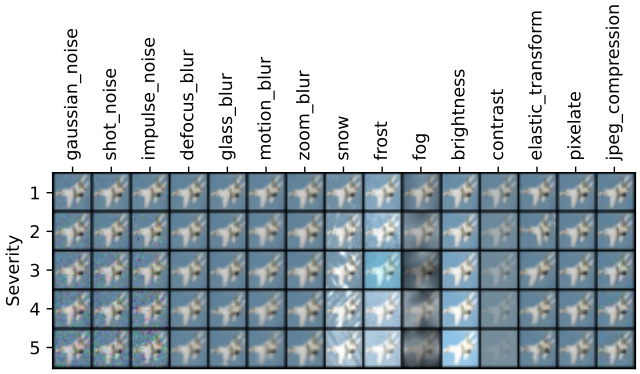


Figure C.11: An example image of CIFAR-10-C with different corruption types at different severity levels. CIFAR-100-C consists of images with the same corruption types and severity levels.

C.9 Main Figures for other Image Datasets

C.9.1 *CIFAR-100 Adversarial Attack Accuracies (Figure 7.2)*

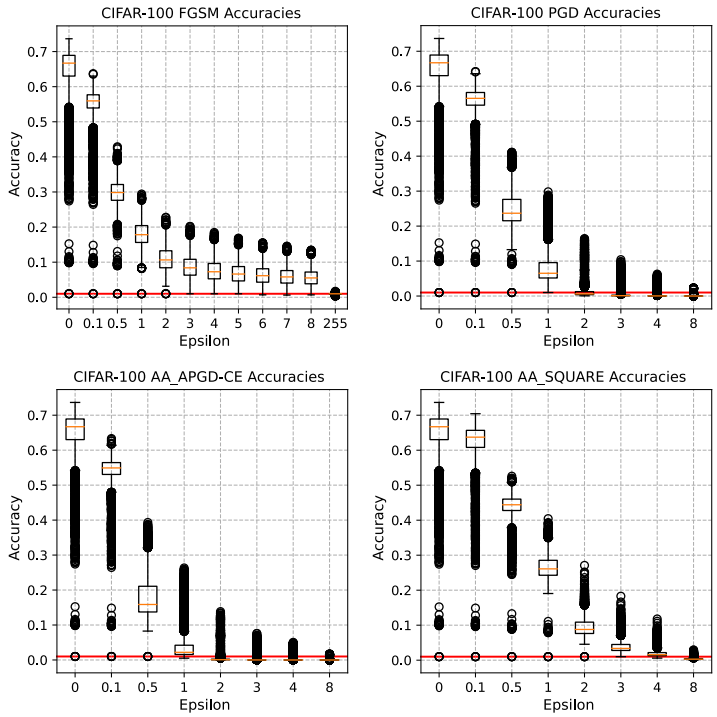


Figure C.12: Accuracy boxplots over all unique architectures in NAS-Bench-201 for different adversarial attacks (FGSM, PGD, APGD, Square) and perturbation magnitude values ϵ, evaluated on CIFAR-100. Red line corresponds to guessing.

C.9.2 *ImageNet16-120 Adversarial Attack Accuracies (Figure 7.2)*

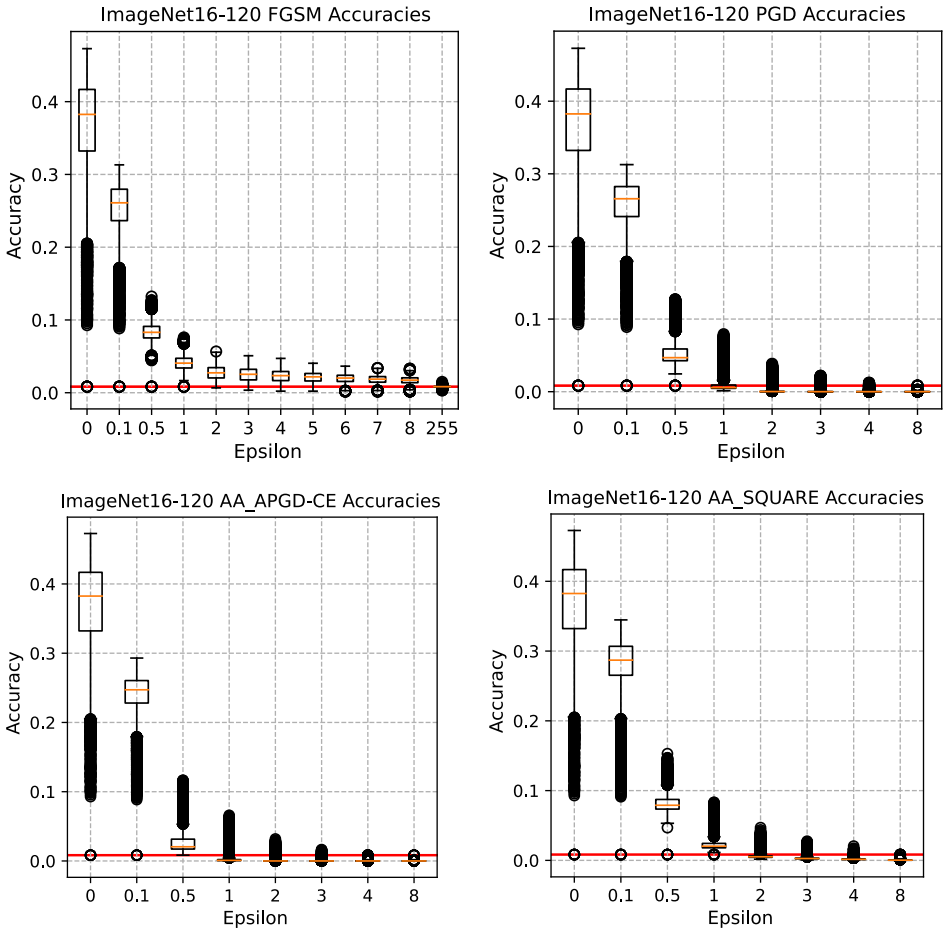


Figure C.13: Accuracy boxplots over all unique architectures in NAS-Bench-201 for different adversarial attacks (FGSM, PGD, APGD, Square) and perturbation magnitude values ϵ, evaluated on ImageNet16-120. Red line corresponds to guessing.

C.9.3 CIFAR-10-C Common Corruption Accuracies (Figure 7.4)

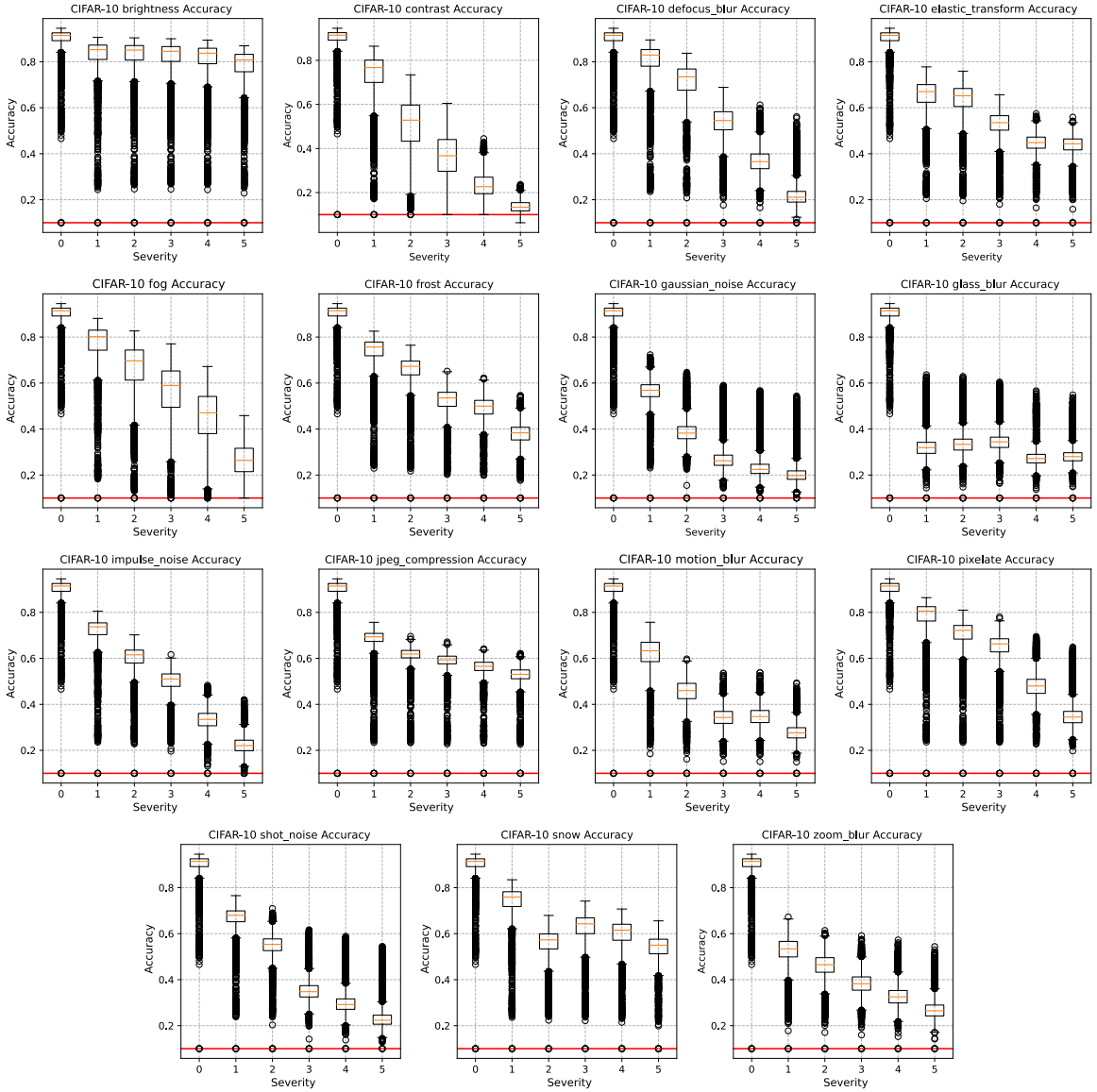


Figure C.14: Accuracy boxplots over all unique architectures in NAS-Bench-201 for different corruption types at different severity levels, evaluated on CIFAR-10-C. Red line corresponds to guessing.

C.9.4 *CIFAR-100-C Common Corruption Accuracies (Figure 7.4)*

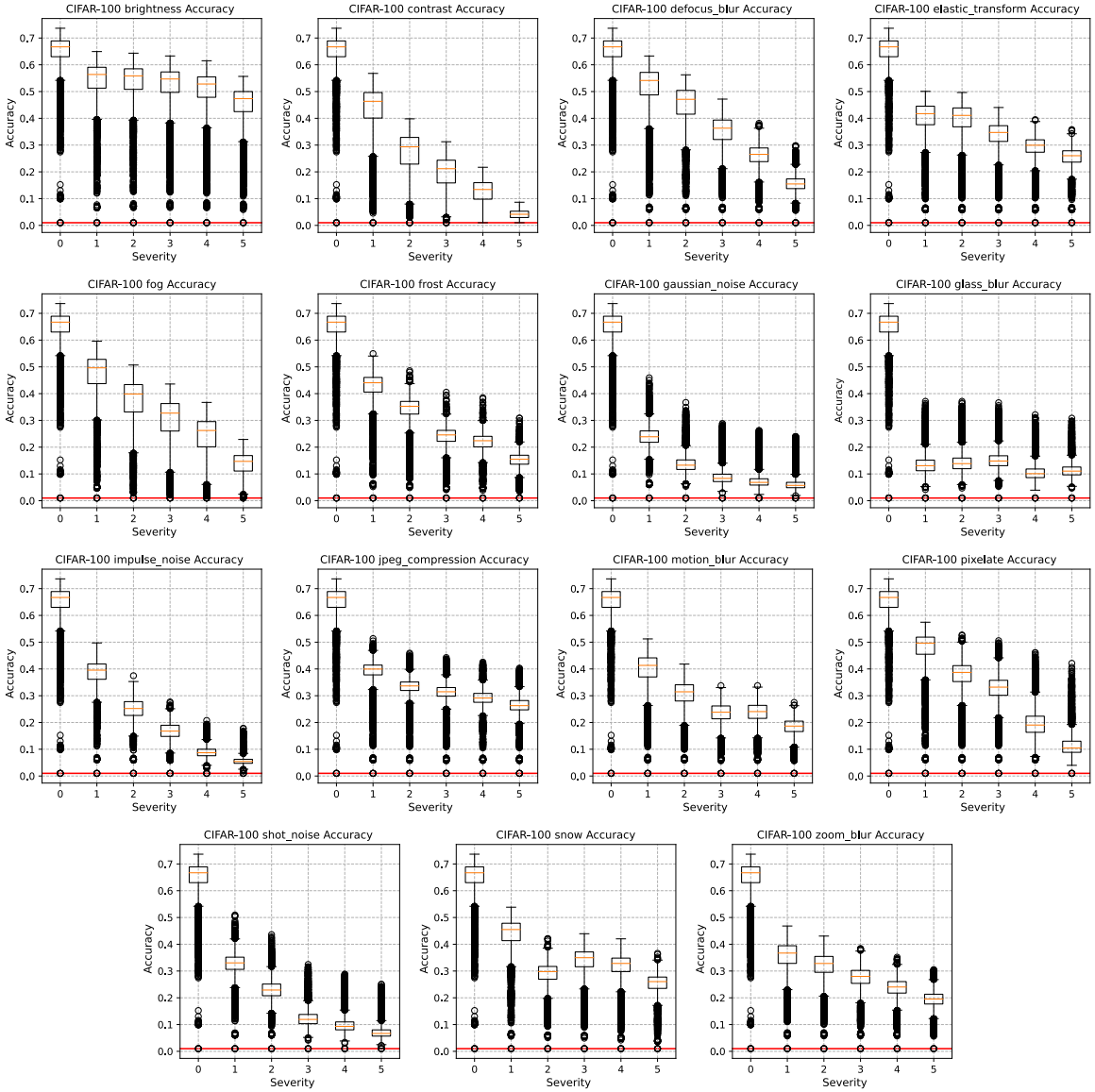


Figure C.15: Accuracy boxplots over all unique architectures in NAS-Bench-201 for different corruption types at different severity levels, evaluated on CIFAR-100-C. Red line corresponds to guessing.

C.9.5 *CIFAR-100 Adversarial Attack Correlations (Figure 7.3)*

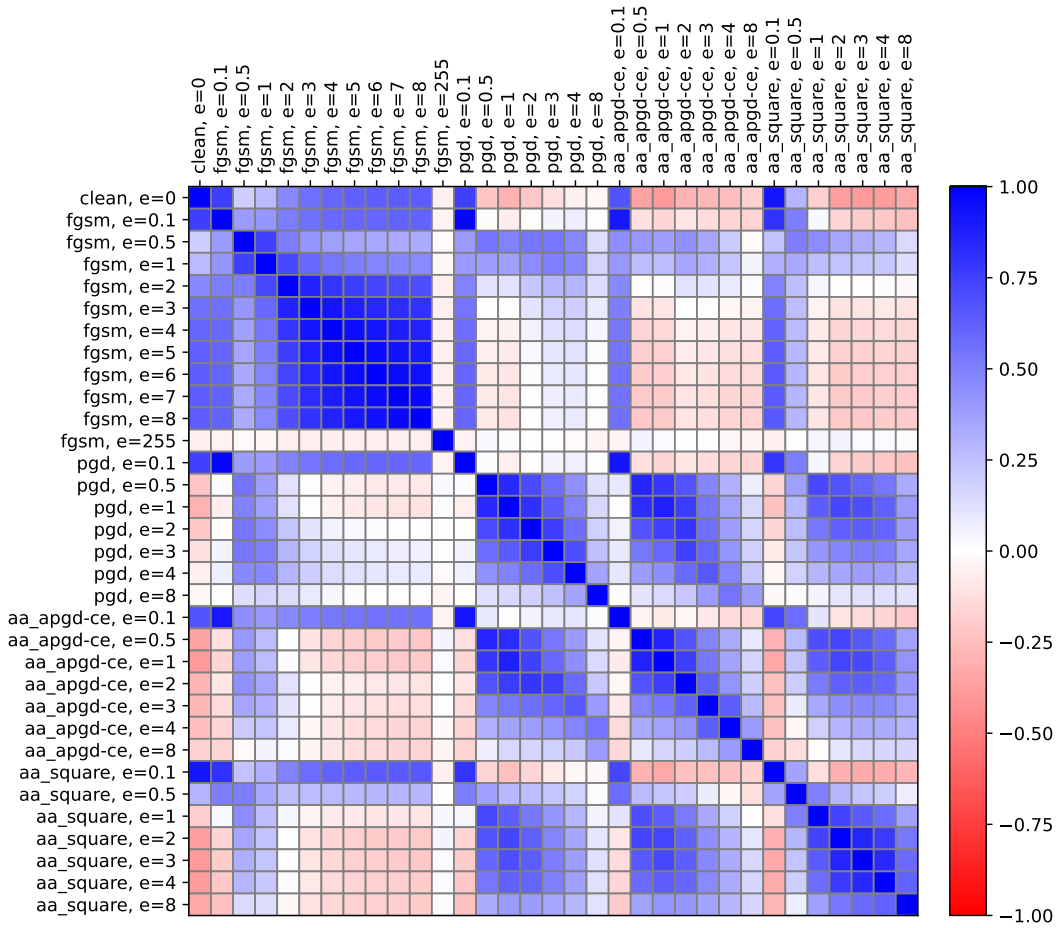


Figure C.16: Kendall rank correlation coefficient between clean accuracies and robust accuracies on different attacks and magnitude values ϵ on CIFAR-100 for all unique architectures in NAS-Bench-201.

C.9.6 *ImageNet16-120 Adversarial Attack Correlations (Figure 7.3)*

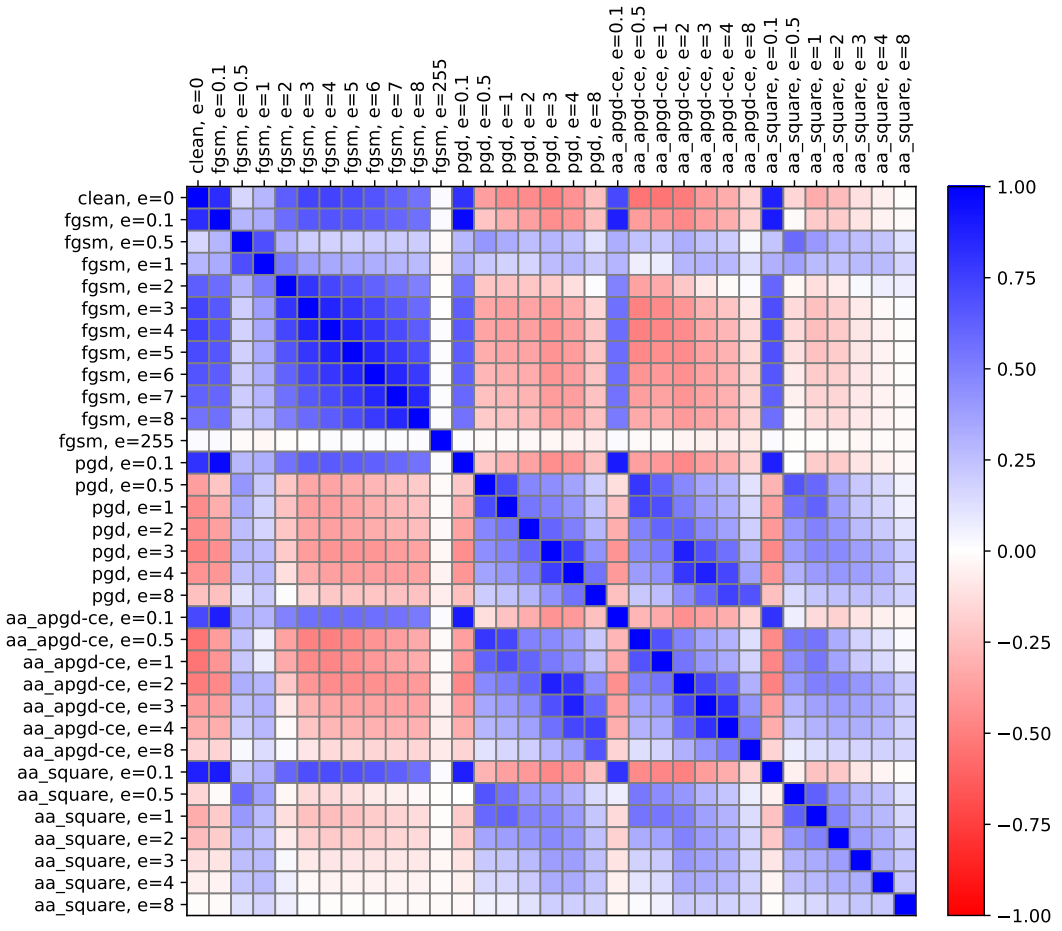


Figure C.17: Kendall rank correlation coefficient between clean accuracies and robust accuracies on different attacks and magnitude values ϵ on ImageNet16-120 for all unique architectures in NAS-Bench-201.

C.9.7 *CIFAR-100-C Common Corruption Correlations (Figure 7.5)*

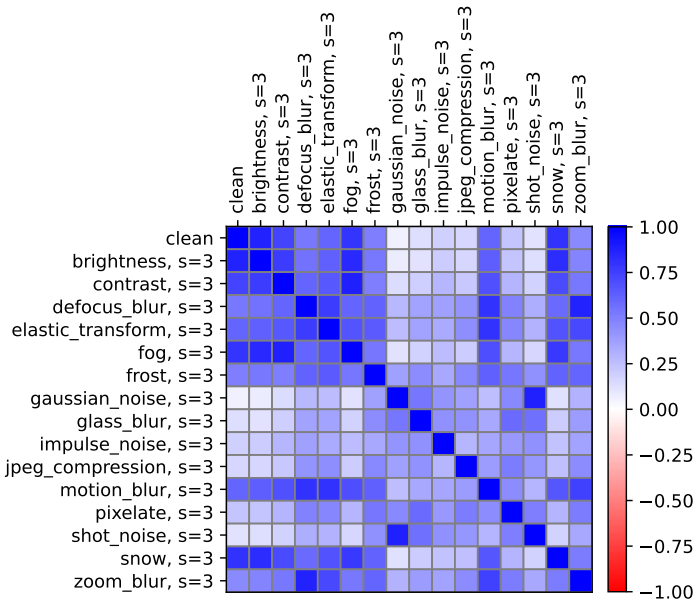


Figure C.18: Kendall rank correlation coefficient between clean accuracies and accuracies on different corruptions at severity level 4 on CIFAR-100-C for all unique architectures in NAS-Bench-201.

Excursus about Clustering on Graphs

In this section we will provide additional information about the optimality of the auxiliary function, and more descriptions about the algorithms used in Chapter 8.

D

D.1 Auxiliary Function at Optimality

In this section, we demonstrate that there exists an $\mathbf{x}^*$, that minimizes Equation (CC_2), for which $Q(\phi, s, \mathbf{x}^*) = 0$. Given an arbitrary solution $\{x_{v_iv_j}, (x^s_{v_iv_j})_{s\in\mathcal{S}}\}$ another solution $\{x^*_{v_iv_j}, (x^{s*}_{v_iv_j})_{s\in\mathcal{S}}\}$ is constructed, for which $Q(\phi, s, \mathbf{x}^*) = 0$ holds, without increasing the objective in Equation (CC_2). We write the updates below in terms of $\mathbf{x}^s$

$$\begin{aligned}
x^*_{v_iv_j} &\triangleq x_{v_iv_j} + \max_{s\in\mathcal{S}} x^s_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^+ \\
x^*_{v_iv_j} &\triangleq x_{v_iv_j} + x^s_{v_iv_j} - 1 \quad \forall (v_i, v_j) \in \mathcal{E}^-_s, s \in \mathcal{S} \\
x^{s*}_{v_iv_j} &\triangleq 0 \quad \forall (v_i, v_j) \in \mathcal{E}^+ \\
x^{s*}_{v_iv_j} &\triangleq 1 \quad \forall (v_i, v_j) \in \mathcal{E}^-_s, s \in \mathcal{S}.
\end{aligned} \tag{D.1}$$

The updates in Equation (D.1) are equivalent to the following updates using f^s, f^{s*}. Here f^s, f^{s*} correspond to the optimizing solution for f in subproblem s, given $\mathbf{x}, \mathbf{x}^*$ respectively:

$$\begin{aligned}
x^*_{v_iv_j} &= x_{v_iv_j} + \max_{s\in\mathcal{S}} f^s_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^+ \\
x^*_{v_iv_j} &= x_{v_iv_j} - f^s_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^-_s, s \in \mathcal{S} \\
f^{s*}_{v_iv_j} &= 0 \quad \forall (v_i, v_j) \in \mathcal{E}^+ \\
f^{s*}_{v_iv_j} &= 0 \quad \forall (v_i, v_j) \in \mathcal{E}^-_s
\end{aligned} \tag{D.2}$$

These updates in Equation (D.1) and Equation (D.2) preserve the feasibility of the primal LP in Equation (8.7). Also notice, that since f^{s*} is a zero valued vector for all $s \in \mathcal{S}$, then $Q(\phi, s, x^*) = 0$ for all $s \in \mathcal{S}$.

We now consider, the total change in Equation (CC_2) corresponding to edge $(v_i, v_j) \in \mathcal{E}^+$, induced by Equation (D.1), which is non-positive. The objective of the master problem increases by $\phi_{v_iv_j} \max_{s\in\mathcal{S}} x^s_{v_iv_j}$, while the total decrease in the objectives of the subproblems is $\phi_{v_iv_j} \sum_{s\in\mathcal{S}} x^s_{v_iv_j}$. Since the latter value is greater than the former value, the total change in problem Equation (CC_2) decreases more than it increases. Considering on the other hand the total change of Equation (CC_2) corresponding to edge $(v_i, v_j) \in \mathcal{E}^-$, induced by Equation (D.1), which is zero, yields in an increase of the objective of the master problem by $-\phi_{v_iv_j}(1 - x^n_{v_iv_j})$, while the objective of subproblem s decreases by $-\phi_{v_iv_j}(1 - x^s_{v_iv_j})$. This shows that the objective of Equation (CC_2) is minimized for $\mathbf{x}^*$.

D.2 Line by Line Description of BDCC

We provide a line by line description of Algorithm 8.

- Input (i) in Algorithm 8: Initialize the nascent set of Benders rows $\widehat{\mathcal{Z}}$ to the empty set.

- Input (ii) in Algorithm 8: Indicate that we have not solved the LP relaxation yet.
- Line 1-15: Alternate between solving the master problem and generating Benders rows, until a feasible integral solution is produced.
  1. Line 2: Solve the master problem providing a solution $\mathbf{x}$, which may not satisfy all cycle inequalities. We enforce integrality if we have finished solving the LP relaxation, which is indicated by done_lp=True.
  2. Line 3: Indicate that we have not yet added any Benders rows to this iteration.
  3. Line 4-11: Add Benders rows by iterating over subproblems and adding Benders rows corresponding to subproblems, associated with violated cycle inequalities.
     - Line 5: Check if there exists a violated cycle inequality associated with $\mathcal{E}_s^-$. This is done by iterating over $(v_i, v_j) \in \mathcal{E}_s^-$ and checking if the shortest path from v_i to v_j is less than $x_{v_i v_j}$. This distance is defined on the graph's edges $\mathcal{E}$ with weights equal to $\mathbf{x}$.
     - Lines 6-8: Generate Benders rows associated with subproblem s and add them to nascent set $\widehat{\mathcal{Z}}$.
     - Line 9: Indicate that a Benders row was added this iteration.
  4. Lines 12-13: If no Benders rows were added to this iteration, we enforce integrality on $\mathbf{x}$, when solving the master problem for the remainder of the algorithm.
- Output in Algorithm 8: Return solution $\mathbf{x}$.

D.3 GENERATING FEASIBLE INTEGER SOLUTIONS PRIOR TO CONVERGENCE

Prior to the termination of optimization, it is valuable to provide feasible integer solutions on demand. This is so that a practitioner can terminate optimization, when the gap between the objectives of the integral solution and the relaxation is small. Here, we consider the production of feasible integer solutions, given the current solution $\mathbf{x}^*$ to the master problem, which may neither obey cycle inequalities nor be integral. We refer to this procedure as rounding.

Rounding is a coordinate descent approach defined on the graph $\mathcal{G}$ and its edges $\mathcal{E}$ with weights κ, determined using $\mathbf{x}^*$ below.

$$\begin{aligned} \kappa_{v_i v_j} &= \phi_{v_i v_j}(1 - x^*_{v_i v_j}) \quad \forall (v_i, v_j) \in \mathcal{E}^+ \\ \kappa_{v_i v_j} &= \phi_{v_i v_j} x^*_{v_i v_j} \quad \forall (v_i, v_j) \in \mathcal{E}^- \end{aligned} \tag{D.3}$$

Consider that $\mathbf{x}^*$ is integral and feasible (where feasibility indicates that $\mathbf{x}^*$ satisfies all cycle inequalities). Let $\mathbf{x}^{s*}$ define the boundaries in partition $\mathbf{x}^*$, of the connected component containing s. Here $x^{s*}_{v_i v_j} = 1$ if exactly one of v_i, v_j is in the connected component containing s under cut $\mathbf{x}^*$. Observe, that $Q(\kappa, s, \mathbf{x}^{0s}) = 0$, where $x^{0s}_{v_i v_j} = \mathbb{1}_{\mathcal{E}_s^-}(v_i, v_j)$, is achieved using $\mathbf{x}^{s*}$ as the solution to Equation (8.7). Thus $\mathbf{x}^{s*}$ is the minimizer of Equation (8.7). The union of the edges cut in $\mathbf{x}^{s*}$ across $s \in \mathcal{S}$ is identical to $\mathbf{x}^*$. Note that when $\mathbf{x}^*$ is integral and feasible then the solution produced below has cost equal to that of $\mathbf{x}^*$.

$$\mathbf{x}^{s*} \triangleq \text{minimizer of } Q(\kappa, s, \mathbf{x}^{0s}) \quad \forall s \in \mathcal{S}$$

Algorithm 10: Generating an Integral and Feasible Solution Given Infeasible and or Non-Integral Input $\mathbf{x}^*$

Input: (i) $x^+_{v_iv_j} = 0 \quad \forall (v_i, v_j) \in \mathcal{E}$
Input: (ii) $\kappa_{v_iv_j} = \phi_{v_iv_j} x^*_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^-$
Input: (iii) $\kappa_{v_iv_j} = \phi_{v_iv_j}(1 - x^*_{v_iv_j}) \quad \forall (v_i, v_j) \in \mathcal{E}^+$
Output: $\mathbf{x}^+$
1 **for** $s \in \mathcal{S}$ **do**
2 $\quad \mathbf{x}^s = $ minimizer for $Q(\kappa, s, \mathbf{x}^{0s})$ given fixed κ, s
3 $\quad x^+_{v_iv_j} = \max(x^+_{v_iv_j}, x^s_{v_iv_j}) \quad \forall (v_i, v_j) \in \mathcal{E}$
4 $\quad \kappa_{v_iv_j} = \kappa_{v_iv_j}(1 - x^+_{v_iv_j}) \quad \forall (v_i, v_j) \in \mathcal{E}$
5 **end**

$$x^+_{v_iv_j} \triangleq \max_{s \in \mathcal{S}} x^{s*}_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^+ \tag{D.4}$$
$$x^+_{v_iv_j} \triangleq x^{s*}_{v_iv_j} \quad \forall (v_i, v_j) \in \mathcal{E}^-_s, s \in \mathcal{S}$$

The procedure of Equation (D.4) can be used regardless of whether $\mathbf{x}^*$ is integral or feasible. Note that if $\mathbf{x}^*$ is close to integral and close to feasible, then Equation (D.4) is biased to produce a solution that is similar to $\mathbf{x}^*$ by design of κ. We now consider a serial version of Equation (D.4), which may provide improved results. We construct a partition $\mathbf{x}^+$ by iterating over $s \in \mathcal{S}$, producing component partitions as in Equation (D.4). We alter κ by allowing for the cutting of edges previously cut with cost zero. We formally describe this serial rounding procedure below in Algorithm 10.

- Input (i) in Algorithm 10: Initialize $\mathbf{x}^+$ as the zero vector.
- Input (ii)- input(iii) in Algorithm 10: Set κ according to Equation (D.3)
- Line 1-5: Iterate over $s \in \mathcal{S}$ to construct $\mathbf{x}^+$ by cutting edges cut in the subproblem.
    1. Line 2: Produce the lowest cost cut $\mathbf{x}^s$ given altered edge weights κ for subproblem s.
    2. Line 3: Cut edges in $\mathbf{x}^+$ that are cut in $\mathbf{x}^s$.
    3. Line 4: Set $\phi_{v_iv_j}$ to zero for cut edges in $\mathbf{x}^+$.
- Output in 10: Return the solution $\mathbf{x}^+$

When solving for the fast minimizer of $Q(\kappa, s, \mathbf{x}^{0s})$, we rely on the network flow solver of Rother et al. (2007), though we do not exploit its capacity to tackle non-submodular problems.

List of Algorithms

1 Bayesian optimization. 23
2 Bayesian optimization in the latent space. 29

3 Two-sided graph generation. 47
4 Decoding of one direction. 48

5 One-sided graph generation. 64
6 Unconstrained search algorithm. 67
7 Constrained search algorithm. 77

8 Benders decomposition for correlation clustering. 128

9 Robustness dataset gathering. 150

10 Generating feasible solution. 169

List of Figures

1.1 ResNet-18 on CIFAR-10. . . . 2
1.2 High-level thesis overview . . . 3
1.3 Overview of architecture latent representation space. . . . 5

2.1 Overview cell-based search space. . . . 13
2.2 Best cells in tabular benchmarks. . . . 18
2.3 Best cells in NAS-Bench-301. . . . 19
2.4 Examples of CIFAR-10. . . . 20
2.5 Parameter Count vs. Test Accuracy. . . . 20

3.1 Graph encoding process. . . . 34
3.2 Performance prediction NAS-Bench-101. . . . 36
3.3 Accuracy distribution in NAS-Bench-101. . . . 37
3.4 GNN performance prediction training progress. . . . 39

4.1 Architecture of proposed variational autoencoder model. . . . 44
4.2 Prediction performance of SVGe. . . . 52
4.3 Architectures on hypersphere circle. . . . 53

5.1 Proposed generative NAS approach. . . . 60
5.2 Training procedure of generator. . . . 63
5.3 Latent space optimization visualization. . . . 66
5.4 Generative search. . . . 71
5.5 Multi-objective search. . . . 74
5.6 Multi-objective search progress. . . . 75
5.7 Prediction model ablation on NAS-Bench-101. . . . 78
5.8 Ablation search progress within the latent space. . . . 80

6.1 Sequential meta-architecture. . . . 88
6.2 Non-sequential meta-architecture. . . . 89
6.3 Scatter plot for blur and downsampling. . . . 93
6.4 Scatter plot for blur and downsampling with optimized hyperparameters. . . . 95
6.5 Scatter plot with improved initialization. . . . 96
6.6 Scatter plot for search of non-sequential meta-architecture with different hyperparameters. . . . 98
6.7 Scatter plot for search of non-sequential meta-architecture with its optimized hyperparameters. . . . 99
6.8 Visualization of found architectures. . . . 100

7.1 Overview NAS-Bench-201 search space. . . . 106
7.2 Adversarial robustness. . . . 107
7.3 Adversarial rank correlation. . . . 108

7.4 Corruption robustness. . . . 109
7.5 Corruption rank correlation. . . . 110
7.6 Rank correlation of training-free robustness measurements. . . . 111
7.7 Best architectures in NAS-Bench-201 according to different accuracies of interest. . . . 114
7.8 Scatter plot robustness accuracies vs. clean accuracies. . . . 114
7.9 Mean robust accuracies vs. kernel parameters. . . . 115
7.10 Comparison top-20 robust architecture. . . . 116
7.11 Scatter plot robustness accuracies vs. clean accuracies and the best performing architecture neighborhood. . . . 117

8.1 Plot of upper and lower bounds on selected problem instances. . . . 129
8.2 Parallelization and MWR. . . . 130

A.1 Cell representation for NAS-Bench-101. . . . 139
A.2 Cell representation for NAS-Bench-201. . . . 140
A.3 Cell representation for NAS-Bench-301. . . . 140
A.4 Cell representation for NAS-Bench-NLP. . . . 141

C.1 Example of two isomorphic graphs in NAS-Bench-201. . . . 149
C.2 Diagram showing the gathering process for our robustness dataset. . . . 151
C.3 Excerpt of meta.json in robustness dataset. . . . 152
C.4 Excerpt of accuracy.json in robustness dataset. . . . 153
C.5 Excerpt of clean_confidence.json in robustness dataset. . . . 154
C.6 Mean confidence scores on CIFAR-10. . . . 155
C.7 Mean label confidence scores on FGSM-attacked CIFAR-10 images. . . . 156
C.8 Mean prediction confidence scores on FGSM-attacked CIFAR-10 images. . . . 157
C.9 Aggregated confusion matrices. . . . 157
C.10 Kendall rank correlation coefficient between clean and adversarial accuracies. . . . 158
C.11 An example image of CIFAR-10-C. . . . 159
C.12 Accuracy boxplots on CIFAR-100. . . . 159
C.13 Accuracy boxplots on ImageNet16-120. . . . 160
C.14 Accuracy boxplots for CIFAR-10-C. . . . 161
C.15 Accuracy boxplots for CIFAR-100-C. . . . 162
C.16 Kendall rank correlation coefficient between clean and robust accuracies on CIFAR-100. 163
C.17 Kendall rank correlation coefficient between clean and robust accuracies on ImageNet16-120. . . . 164
C.18 Kendall rank correlation coefficient between clean and corruptions accuracies on CIFAR-100-C at severity 4. . . . 165

List of Tables

3.1 Performance prediction on NAS-Bench-101. . . . 36
3.2 Zero-Shot performance prediction on NAS-Bench-101. . . . 37
3.3 Performance comparison for different training data sizes in NAS-Bench-101. . . . 39

4.1 Variational autoencoder abilities. . . . 51
4.2 Performance prediction with different training data sizes. . . . 53
4.3 Architecture Search on NAS-Bench-101. . . . 54
4.4 Architecture extrapolation. . . . 56
4.5 Dataset transfer learning to ImageNet16-120. . . . 56

5.1 Comparison search results on NAS-Bench-101. . . . 68
5.2 Comparison search results on NAS-Bench-201. . . . 69
5.3 Comparison search results on surrogate benchmarks. . . . 70
5.4 Comparison search results on DARTS for ImageNet. . . . 72
5.5 Search results on multi-objective target. . . . 73
5.6 AG-Net generator abilities. . . . 78
5.7 Ablation on latent space optimization. . . . 78
5.8 Ablation search results within the latent space. . . . 79

6.1 Architecture validation PSNR values for 1D inverse problems. . . . 92
6.2 Architecture validation PSNR values for 1D inverse problems with different hyperparameter. . . . 94
6.3 Architecture validation PSNR values for 1D inverse problems for the non-sequential search space. . . . 97
6.4 Architecture validation PSNR values for 1D inverse problems in the non-sequential search space. . . . 99

7.1 Neural architecture search for robustness performance. . . . 113

8.1 Percentage of problems solved that have a duality gap. . . . 130

B.1 Hyperparameters of the GNN surrogate model. . . . 143
B.2 Hyperparameters of the autoencoder model. . . . 144
B.3 Hyperparameters of the latent space surrogate model. . . . 144
B.4 Hyperparameters of the generator model. . . . 145
B.5 Hyperparameters for the performance surrogate model $f(\cdot)$. . . . 145
B.6 Hyperparameters for both surrogate models for the multi-objective search. . . . 146
B.7 General hyperparameters for DAS. . . . 146
B.8 Manually chosen hyperparameters H1 and H2. . . . 146
B.9 BOHB optimized hyperparameters for the sequential search space in DAS. . . . 147
B.10 BOHB optimized hyperparameters for the non-sequential search space in DAS. . . . 147

C.1 Hyperparameter settings of adversarial attacks evaluated. . . . 150

C.2 Keys for attacks and corruptions evaluated. 152
C.3 Files and their possible content. 153

Bibliography

Adler, J. and O. Öktem (2018). “Learned Primal-Dual Reconstruction”. In: *IEEE Transactions on Medical Imaging* 37.6, pp. 1322–1332.

Aggarwal, H. K., M. P. Mani, and M. Jacob (2019). “MoDL: Model-Based Deep Learning Architecture for Inverse Problems”. In: *IEEE Transactions on Medical Imaging* 38.2, pp. 394–405.

Akimoto, Y., S. Shirakawa, N. Yoshinari, K. Uchida, S. Saito, and K. Nishida (2019). “Adaptive Stochastic Natural Gradient Method for One-Shot Neural Architecture Search”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Andres, B., B. T., and J. H. Kappes (2012a). “OpenGM: A C++ Library for Discrete Graphical Models”. In: *arXiv.org* abs/1206.0111.

Andres, B., J. H. Kappes, T. Beier, U. Kothe, and F. A. Hamprecht (2011). “Probabilistic image segmentation with closedness constraints”. In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

Andres, B., T. Kroger, K. L. Briggman, W. Denk, N. Korogod, G. Knott, U. Kothe, and F. A. Hamprecht (2012b). “Globally optimal closed-surface segmentation for connectomics”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Andres, B., J. Yarkony, B. S. Manjunath, S. Kirchhoff, E. Turetken, C. Fowlkes, and H. Pfister. (2013). “Segmenting planar superpixel adjacency graphs w.r.t. non-planar superpixel affinity graphs.” In: *Proc. of the Conference on Energy Minimization in Computer Vision and Pattern Recognition*.

Andriushchenko, M., F. Croce, N. Flammarion, and M. Hein (2020). “Square attack: a query-efficient black-box adversarial attack via random search”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Baker, B., O. Gupta, N. Naik, and R. Raskar (2017). “Designing Neural Network Architectures using Reinforcement Learning”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Baker, B., O. Gupta, R. Raskar, and N. Naik (2018). “Accelerating Neural Architecture Search using Performance Prediction”. In: *Proc. of the International Conference on Learning Representations (ICLR) Workshops*.

Bansal, N., A. Blum, and S. Chawla (2004). “Correlation Clustering”. In: *Machine Learning* 56.1-3, pp. 89–113.

Barnhart, C., E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance (1996). “Branch-and-Price: Column Generation for Solving Huge Integer Programs”. In: *Operations Research* 46.3, pp. 316–329.

Bastings, J., I. Titov, W. Aziz, D. Marcheggiani, and K. Sima’an (2017). “Graph Convolutional Encoders for Syntax-aware Neural Machine Translation”. In: *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Battaglia, P. W., J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. (2018). “Relational inductive biases, deep learning, and graph networks”. In: *arXiv.org* abs/1806.01261.

Beier, T., B. Andres, K. Ullrich, and F. A. Hamprecht (2016). “An Efficient Fusion Move Algorithm for the Minimum Cost Lifted Multicut Problem”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Beier, T., F. A. Hamprecht, and J. H. Kappes (2015). “Fusion moves for correlation clustering”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Beier, T., T. Kroeger, J. H. Kappes, U. Kothe, and F. A. Hamprecht (2014). “Cut, Glue, & Cut: A Fast, Approximate Solver for Multicut Partitioning”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Bender, G., P. Kindermans, B. Zoph, V. Vasudevan, and Q. V. Le (2018). “Understanding and Simplifying One-Shot Architecture Search”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Benders, J. F. (1962). “Partitioning procedures for solving mixed-variables programming problems”. In: *Numerische mathematik* 4.1, pp. 238–252.

Birge, J. R. (1985). “Decomposition and partitioning methods for multistage stochastic linear programs”. In: *Operations Research* 33.5, pp. 989–1007.

Blei, D. M., A. Kucukelbir, and J. D. McAuliffe (2017). “Variational Inference: A Review for Statisticians”. In: *Journal of the American Statistical Association* 112.518, pp. 859–877.

Brochu, E., V. M. Cora, and N. de Freitas (2010). “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning”. In: *arXiv.org* abs/1012.2599.

Bronstein, M. M., J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst (2017). “Geometric deep learning: going beyond euclidean data”. In: *IEEE Signal Processing Magazine* 34.4, pp. 18–42.

Bruna, J., W. Zaremba, A. Szlam, and Y. LeCun (2014). “Spectral Networks and Locally Connected Networks on Graphs”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Cai, H., L. Zhu, and S. Han (2019). “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Carlini, N., A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin (2019). “On Evaluating Adversarial Robustness”. In: *arXiv.org* abs/1902.06705.

Chatzimichailidis, A., J. Keuper, F. Pfreundt, and N. R. Gauger (2019). “GradVis: Visualization and Second Order Analysis of Optimization Surfaces during the Training of Deep Neural Networks”. In: *Proc. of the IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*.

Chen, T. and C. Guestrin (2016). “XGBoost: A Scalable Tree Boosting System”. In: *Proc. of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*.

Chen, W., X. Gong, and Z. Wang (2021). “Neural Architecture Search on ImageNet in Four GPU Hours: A Theoretically Inspired Perspective”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Chen, X. and C. Hsieh (2020). “Stabilizing Differentiable Architecture Search via Perturbation-based Regularization”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Chen, X., L. Xie, J. Wu, and Q. Tian (2019). “Progressive Differentiable Architecture Search: Bridging the Depth Gap Between Search and Evaluation”. In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

Chen, Y., T. Krishna, J. S. Emer, and V. Sze (2017). “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE Journal of Solid State Circuits* 52.1, pp. 127–138.

Cho, K., B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine

Translation". In: *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Chrabaszcz, P., I. Loshchilov, and F. Hutter (2017). "A Downsampled Variant of ImageNet as an Alternative to the CIFAR datasets". In: *arXiv.org* abs/1707.08819.

Chu, X., T. Zhou, B. Zhang, and J. Li (2020). "Fair DARTS: Eliminating Unfair Advantages in Differentiable Architecture Search". In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Cordeau, J.-F., G. Stojković, F. Soumis, and J. Desrosiers (2001). "Benders decomposition for simultaneous aircraft routing and crew scheduling". In: *Transportation Sience* 35.4, pp. 375–388.

Cox, D. and S. John (1992). "A statistical method for global optimization". In: *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*.

Croce, F., M. Andriushchenko, V. Sehwag, E. Debenedetti, N. Flammarion, M. Chiang, P. Mittal, and M. Hein (2021). "RobustBench: a standardized adversarial robustness benchmark". In: *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.

Croce, F. and M. Hein (2020). "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks". In: *Proc. of the International Conference on Machine Learning (ICML)*.

Dai, X., A. Wan, P. Zhang, B. Wu, Z. He, Z. Wei, K. Chen, Y. Tian, M. Yu, P. Vajda, and J. E. Gonzalez (2021). "FBNetV3: Joint Architecture-Recipe Search Using Predictor Pretraining". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). "ImageNet: A large-scale hierarchical image database". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Devaguptapu, C., D. Agarwal, G. Mittal, P. Gopalani, and V. N. Balasubramanian (2021). "On Adversarial Robustness: A Neural Architecture Search perspective". In: *Proc. of the IEEE International Conference on Computer Vision (ICCV) Workshops*.

Dodge, S. F. and L. J. Karam (2017). "A Study and Comparison of Human and Deep Learning Recognition Performance under Visual Distortions". In: *International Conference on Computer Communication and Networks (ICCCN)*.

Domhan, T., J. T. Springenberg, and F. Hutter (2015). "Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves". In: *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*.

Dong, M., Y. Li, Y. Wang, and C. Xu (2020a). "Adversarially Robust Neural Architectures". In: *arXiv.org* abs/2009.00902.

Dong, X. and Y. Yang (2019a). "One-Shot Neural Architecture Search via Self-Evaluated Template Network". In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

— (2019b). "Searching for a Robust Neural Architecture in Four GPU Hours". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

— (2020). "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search". In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Dong, Y., Q.-A. Fu, X. Yang, T. Pang, H. Su, Z. Xiao, and J. Zhu (2020b). "Benchmarking Adversarial Robustness on Image Classification". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Duan, Y., X. Chen, H. Xu, Z. Chen, X. Liang, T. Zhang, and Z. Li (2021). “TransNAS-Bench-101: Improving Transferability and Generalizability of Cross-Task Neural Architecture Search”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Elsken, T., J. H. Metzen, and F. Hutter (2019). “Neural Architecture Search: A Survey”. In: *Journal of Machine Learning Research* 20, 55:1–55:21.

Falkner, S., A. Klein, and F. Hutter (2018). “BOHB: Robust and Efficient Hyperparameter Optimization at Scale”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Fey, M. and J. E. Lenssen (2019). “Fast Graph Representation Learning with PyTorch Geometric”. In: *Proc. of the International Conference on Learning Representations (ICLR), Workshop on Representation Learning on Graphs and Manifolds*.

Fisher, M. E. (1966). “On the Dimer Solution of Planar Ising Models”. In: *Journal of Mathematical Physics* 7.10, pp. 1776–1781.

Ford, L. R. and D. R. Fulkerson (1956). “Maximal flow through a network”. In: *Canadian journal of Mathematics* 8.3, pp. 399–404.

Geiping, J., J. Lukasik, M. Keuper, and M. Moeller (2021a). “DARTS for Inverse Problems: a Study on Stability”. In: *Advances in Neural Information Processing Systems (NeurIPS), Workshop on Deep Learning and Inverse Problems*.

— (2021b). “Is Differentiable Architecture Search truly a One-Shot Method?” In: *arXiv.org* abs/2108.05647.

Geirhos, R., P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel (2019). “ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Geoffrion, A. M. and G. W. Graves (1974). “Multicommodity distribution system design by Benders decomposition”. In: *Management Science* 20.5, pp. 822–844.

Gilmer, J., S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl (2017). “Neural message passing for quantum chemistry”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Gilmore, P. C. and R. E. Gomory (1961). “A Linear Programming Approach to the Cutting-Stock Problem”. In: *Operations Research* 9.6, pp. 849–859.

Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio (2014). “Generative adversarial nets”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Goodfellow, I. J., J. Shlens, and C. Szegedy (2015). “Explaining and Harnessing Adversarial Examples”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Google LLC. Edge TPU Compiler (). Accessed: 2021-11-17.

Google LLC. Pixel 3 (). Accessed: 2021-11-17.

Gori, M., G. Monfardini, and F. Scarselli (2005). “A new model for learning in graph domains”. In: *International Joint Conference on Neural Networks (IJCNN)*.

Gregor, K. and Y. LeCun (2010). “Learning Fast Approximations of Sparse Coding”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Guo, M., Y. Yang, R. Xu, Z. Liu, and D. Lin (2020a). “When NAS Meets Robustness: In Search of Robust Architectures Against Adversarial Attacks”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Guo, Z., X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun (2020b). “Single Path One-Shot Neural Architecture Search with Uniform Sampling”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Gómez-Bombarelli, R., J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik (2018). "Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules". In: *ACS Central Science* 4.2, pp. 268–276.

Hamilton, W., Z. Ying, and J. Leskovec (2017). "Inductive representation learning on large graphs". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Hammernik, K., T. Klatzer, E. Kobler, M. P. Recht, D. K. Sodickson, T. Pock, and F. Knoll (2018). "Learning a Variational Network for Reconstruction of Accelerated MRI Data". In: *Magnetic Resonance in Medicine* 79.6, pp. 3055–3071.

Hammernik, K., T. Würfl, T. Pock, and A. Maier (2017). "A Deep Learning Architecture for Limited-Angle Computed Tomography Reconstruction". In: *Proc. of Workshop Bildverarbeitung für die Medizin*. Informatik aktuell, pp. 92–97.

He, C., H. Ye, L. Shen, and T. Zhang (2020). "MiLeNAS: Efficient Neural Architecture Search via Mixed-Level Reformulation". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

He, K., X. Zhang, S. Ren, and J. Sun (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

— (2016). "Deep residual learning for image recognition". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Henaff, M., J. Bruna, and Y. LeCun (2015). "Deep convolutional networks on graph-structured data". In: *arXiv.org* abs/1506.05163.

Hendrycks, D. and T. Dietterich (2019). "Benchmarking Neural Network Robustness to Common Corruptions and Perturbations". In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Hoffman, J., D. A. Roberts, and S. Yaida (2019). "Robust Learning with Jacobian Regularization". In: *arXiv.org* abs/1908.02729.

Hosseini, R., X. Yang, and P. Xie (2021). "DSRNA: Differentiable Search of Robust Neural Architectures". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Huang, S. and W. Chu (2021). "Searching by Generating: Flexible and Efficient One-Shot NAS With Architecture Generator". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Insafutdinov, E., L. Pishchulin, B. Andres, M. Andriluka, and B. Schiele (2016). "Deepercut: A deeper, stronger, and faster multi-person pose estimation model". In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Jin, W., R. Barzilay, and T. S. Jaakkola (2018). "Junction Tree Variational Autoencoder for Molecular Graph Generation". In: *Proc. of the International Conference on Machine Learning (ICML)*.

Jones, D. R., M. Schonlau, and W. J. Welch (1998). "Efficient Global Optimization of Expensive Black-Box Functions". In: *Journal of Global Optimization* 13.4, pp. 455–492.

Jung, S., J. Lukasik, and M. Keuper (2023). "Neural Architecture Design and Robustness: A Dataset". In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Kandasamy, K., W. Neiswanger, J. Schneider, B. Póczos, and E. P. Xing (2018). "Neural Architecture Search with Bayesian Optimisation and Optimal Transport". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Kardoost, A. and M. Keuper (2018). “Solving Minimum Cost Lifted Multicut Problems by Node Agglomeration”. In: *Proc. of the Asian Conference on Computer Vision (ACCV)*.

Keuper, M., B. Andres, and T. Brox (2015a). “Motion Trajectory Segmentation via Minimum Cost Multicuts”. In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

Keuper, M., E. Levinkov, N. Bonneel, G. Lavoué, T. Brox, and B. Andres (2015b). “Efficient Decomposition of Image and Mesh Graphs by Lifted Multicuts”. In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

Kim, S., S. Nowozin, P. Kohli, and C. D. Yoo (2011). “Higher-Order Correlation Clustering for Image Segmentation”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Kingma, D. P. and J. Ba (2015). “Adam: A Method for Stochastic Optimization”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Kingma, D. P. and M. Welling (2014). “Auto-Encoding Variational Bayes”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Kipf, T. N. and M. Welling (2016). “Variational graph auto-encoders”. In: *arXiv.org* abs/1611.07308.

Kipf, T. N. and M. Welling (2017). “Semi-Supervised Classification with Graph Convolutional Networks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Klatzer, T., K. Hammernik, P. Knobelreiter, and T. Pock (2016). “Learning Joint Demosaicing and Denoising Based on Sequential Energy Minimization”. In: *Proc. of the IEEE International Conference on Computational Photography (ICCP)*.

Klein, A., S. Falkner, J. T. Springenberg, and F. Hutter (2017). “Learning Curve Prediction with Bayesian Neural Networks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Klyuchnikov, N., I. Trofimov, E. Artemova, M. Salnikov, M. V. Fedorov, A. Filippov, and E. Burnaev (2022). “NAS-Bench-NLP: Neural Architecture Search Benchmark for Natural Language Processing”. In: *IEEE Access* 10, pp. 45736–45747.

Kobler, E., T. Klatzer, K. Hammernik, and T. Pock (2017). “Variational Networks: Connecting Variational Methods and Deep Learning”. In: *Proc. of the German Conference on Pattern Recognition (GCPR)*.

Kolmogorov, V. (2009). “Blossom V: a new implementation of a minimum cost perfect matching algorithm”. In: *Mathematical Programming Computation* 1.1, pp. 43–67.

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. University of Toronto.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Kurakin, A., I. J. Goodfellow, and S. Bengio (2017). “Adversarial Machine Learning at Scale”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Kusner, M. J., B. Paige, and J. M. Hernández-Lobato (2017). “Grammar Variational Autoencoder”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Landrieu, L. and M. Simonovsky (2018). “Large-scale point cloud semantic segmentation with superpoint graphs”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Li, C., Z. Yu, Y. Fu, Y. Zhang, Y. Zhao, H. You, Q. Yu, Y. Wang, C. Hao, and Y. Lin (2021a). “HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Li, G., G. Qian, I. C. Delgadillo, M. Müller, A. K. Thabet, and B. Ghanem (2020). “SGAS: Sequential Greedy Architecture Search”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Li, L., M. Khodak, N. Balcan, and A. Talwalkar (2021b). “Geometry-Aware Gradient Algorithms for Neural Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Li, L. and A. Talwalkar (2019). “Random Search and Reproducibility for Neural Architecture Search”. In: *Proc. of the Conference on Uncertainty in Artificial Intelligence (UAI)*.

Li, L., K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar (2018a). “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *Journal of Machine Learning Research* 18.185, pp. 1–52.

Li, Y., D. Tarlow, M. Brockschmidt, and R. S. Zemel (2016). “Gated Graph Sequence Neural Networks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Li, Y., O. Vinyals, C. Dyer, R. Pascanu, and P. W. Battaglia (2018b). “Learning Deep Generative Models of Graphs”. In: *arXiv.org* abs/1803.03324.

Lindauer, M. and F. Hutter (2020). “Best Practices for Scientific Research on Neural Architecture Search”. In: *Journal of Machine Learning Research* 21.243, pp. 1–18.

Ling, X., S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang (2019). “DEEPSEC: A Uniform Platform for Security Analysis of Deep Learning Model”. In: *Proc. of the IEEE Symposium on Security and Privacy*.

Liu, C., B. Zoph, M. Neumann, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A. L. Yuille, J. Huang, and K. Murphy (2018a). “Progressive Neural Architecture Search”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Liu, H., K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu (2018b). “Hierarchical Representations for Efficient Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Liu, H., K. Simonyan, and Y. Yang (2019). “DARTS: Differentiable Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Long, D., S. Zhang, and Y. Zhang (2020). “Performance Prediction Based on Neural Architecture Features”. In: *Cognitive Computation and Systems* 2.2, pp. 80–83.

Lukasik, J., D. Friede, H. Stuckenschmidt, and M. Keuper (2020a). “Neural Architecture Performance Prediction Using Graph Neural Networks”. In: *Proc. of the German Conference on Pattern Recognition (GCPR)*.

Lukasik, J., D. Friede, A. Zela, F. Hutter, and M. Keuper (2021). “Smooth Variational Graph Embeddings for Efficient Neural Architecture Search”. In: *International Joint Conference on Neural Networks (IJCNN)*.

Lukasik, J., S. Jung, and M. Keuper (2022). “Learning Where to Look - Generative NAS is Surprisingly Efficient”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Lukasik, J., M. Keuper, M. Singh, and J. Yarkony (2020b). “A Benders Decomposition Approach to Correlation Clustering”. In: *Proc. of the IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*.

Luo, R., F. Tian, T. Qin, E. Chen, and T. Liu (2018). “Neural Architecture Optimization”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Magnanti, T. L. and R. T. Wong (1981). “Accelerating Benders decomposition: Algorithmic enhancement and model selection criteria”. In: *Operations Research* 29.3, pp. 464–484.

Martin, D., C. Fowlkes, D. Tal, and J. Malik (2001). “A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics”. In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

Mehrotra, A., A. G. C. P. Ramos, S. Bhattacharya, L. Dudziak, R. Vipperla, T. Chau, M. S. Abdelfattah, S. Ishtiaq, and N. D. Lane (2021). “NAS-Bench-ASR: Reproducible Neural Architecture Search for Speech Recognition”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Mehta, Y., C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter (2022). “NAS-Bench-Suite: NAS Evaluation is (Now) Surprisingly Easy”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Mellor, J., J. Turner, A. J. Storkey, and E. J. Crowley (2021). “Neural Architecture Search without Training”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Mikolov, T., M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur (2010). “Recurrent neural network based language model”. In: *Annual Conference of the International Speech Communication Association*.

Mockus, J., V. Tiesis, and A. Zilinskas (1978). “The application of Bayesian methods for seeking the extremum”. In: *Towards Global Optimization*. Vol. 2, pp. 117–129.

Mockus, J. (1974). “On Bayesian Methods for Seeking the Extremum”. In: *Optimization Techniques*, pp. 400–404.

Mok, J., B. Na, H. Choe, and S. Yoon (2021). “AdvRush: Searching for Adversarially Robust Neural Architectures”. In: *Proc. of the IEEE International Conference on Computer Vision (ICCV)*.

Monti, F., M. Bronstein, and X. Bresson (2017). “Geometric matrix completion with recurrent multi-graph neural networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Niepert, M., M. Ahmed, and K. Kutzkov (2016). “Learning convolutional neural networks for graphs”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Ning, X., Y. Zheng, T. Zhao, Y. Wang, and H. Yang (2020). “A Generic Graph-Based Neural Architecture Encoding Scheme for Predictor-Based NAS”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Nowozin, S. and S. Jegelka (2009). “Solution stability in linear programming relaxations: Graph partitioning and unsupervised learning”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

NVIDIA Jetson TX2 (). Accessed: 2021-11-17.

Osborne, M. A., R. Garnett, and S. J. Roberts (2008). *Gaussian processes for Global Optimization*. Tech. rep. University of Oxford.

Pan, X., D. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan (2015). “Parallel Correlation Clustering on Big Graphs”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Pang, T., X. Yang, Y. Dong, H. Su, and J. Zhu (2021). “Bag of Tricks for Adversarial Training”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Pham, H., M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean (2018). “Efficient Neural Architecture Search via Parameter Sharing”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Pishchulin, L., E. Insafutdinov, S. Tang, B. Andres, M. Andriluka, P. V. Gehler, and B. Schiele (2016). “Deepcut: Joint subset partition and labeling for multi person pose estimation”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Radosavovic, I., R. P. Kosaraju, R. B. Girshick, K. He, and P. Dollár (2020). “Designing Network Design Spaces”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Rasmussen, C. E. and C. K. I. Williams (2006). *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press.

Raspberry Pi Limited (). Accessed: 2021-11-17.

Rauber, J., W. Brendel, and M. Bethge (2017). “Foolbox: A Python toolbox to benchmark the robustness of machine learning models”. In: *Proc. of the International Conference on Machine Learning (ICML), Reliable Machine Learning in the Wild Workshop*.

Real, E., A. Aggarwal, Y. Huang, and Q. V. Le (2019). “Regularized Evolution for Image Classifier Architecture Search”. In: *Proc. of the Conference of Artificial Intelligence (AAAI)*.

Real, E., S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin (2017). “Large-Scale Evolution of Image Classifiers”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Rezaei, S. S. C., F. X. Han, D. Niu, M. Salameh, K. G. Mills, S. Lian, W. Lu, and S. Jui (2021). “Generative Adversarial Neural Architecture Search”. In: *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*.

Riegler, G., M. Rüther, and H. Bischof (2016). “Atgv-Net: Accurate Depth Super-Resolution”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Roberts, N., M. Khodak, T. Dao, L. Li, C. Ré, and A. Talwalkar (2021). “Rethinking Neural Operations for Diverse Tasks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Rolínek, M., V. Musil, A. Paulus, M. V. P., C. Michaelis, and G. Martius (2020). “Optimizing Rank-Based Metrics With Blackbox Differentiation”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Rother, C., V. Kolmogorov, V. Lempitsky, and M. Szummer (2007). “Optimizing Binary MRFs via Extended Roof Duality”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ru, B., X. Wan, X. Dong, and M. Osborne (2021). “Interpretable Neural Architecture Search via Bayesian Optimisation with Weisfeiler-Lehman Kernels”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini (2009). “The Graph Neural Network Model”. In: *IEEE Trans. Neural Networks* 20.1, pp. 61–80.

Schmidt, U. and S. Roth (2014). “Shrinkage Fields for Effective Image Restoration”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Shahriari, B., K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas (2016). “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proc. of the IEEE* 104.1, pp. 148–175.

Shi, H., R. Pi, H. Xu, Z. Li, J. T. Kwok, and T. Zhang (2019). “Multi-objective Neural Architecture Search via Predictive Network Performance Optimization”. In: *arXiv.org* abs/1911.09336.

Shih, W.-K., S. Wu, and Y. Kuo (1990). “Unifying maximum cut and minimum cut of a planar graph”. In: *IEEE Transactions on Computers* 39.5, pp. 694–697.

Simonovsky, M. and N. Komodakis (2018). “Graphvae: Towards generation of small graphs using variational autoencoders”. In: *International Conference on Artificial Neural Networks (ICANN)*.

Simonyan, K. and A. Zisserman (2014). “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Snelson, E. and Z. Ghahramani (2005). “Sparse Gaussian Processes using Pseudo-inputs”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Snoek, J., O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams (2015). “Scalable Bayesian Optimization Using Deep Neural Networks”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Su, D., H. Zhang, H. Chen, J. Yi, P.-Y. Chen, and Y. Gao (2018). “Is Robustness the Cost of Accuracy? - A Comprehensive Study on the Robustness of 18 Deep Image Classification Models”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Sullivan, T. (2015). *Introduction to Uncertainty Quantification*.

Swoboda, P. and B. Andres (2017). “A Message Passing Algorithm for the Minimum Cost Multicut Problem”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). “Going deeper with convolutions”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Szegedy, C., V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna (2016). “Rethinking the Inception Architecture for Computer Vision”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus (2014). “Intriguing properties of neural networks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Tang, S., B. Andres, M. Andriluka, and B. Schiele (2015). “Subgraph Decomposition for Multi-Target Tracking”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Tang, Y., Y. Wang, Y. Xu, H. Chen, B. Shi, C. Xu, C. Xu, Q. Tian, and C. Xu (2020). “A Semi-Supervised Assessor of Neural Architectures”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Titsias, M. (2009). “Variational Learning of Inducing Variables in Sparse Gaussian Processes”. In: *Proc. of the International Conference on Artificial Intelligence and Statistics*.

Tripp, A., E. Daxberger, and J. M. Hernández-Lobato (2020). “Sample-Efficient Optimization in the Latent Space of Deep Generative Models via Weighted Retraining”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Tu, R., N. Roberts, M. Khodak, J. Shen, F. Sala, and A. Talwalkar (2022). “NAS-Bench-360: Benchmarking Neural Architecture Search on Diverse Tasks”. In: *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.

Velickovic, P., G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio (2018). “Graph Attention Networks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Vicol, P., J. P. Lorraine, F. Pedregosa, D. Duvenaud, and R. B. Grosse (2022). “On Implicit Bias in Overparameterized Bilevel Optimization”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Wan, X., B. Ru, P. M. Esperança, and Z. Li (2022). “On Redundancy and Diversity in Cell-based Neural Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Wang, S., A. Ihler, K. Kording, and J. Yarkony (2018). “Accelerating Dynamic Programs via Nested Benders Decomposition with Application to Multi-Person Pose Estimation”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Wang, S., K. Kording, and J. Yarkony (2017). “Exploiting skeletal structure in computer vision annotation with Benders decomposition”. In: *arXiv.org* abs/1709.04411.

Wen, W., H. Liu, Y. Chen, H. H. Li, G. Bender, and P. Kindermans (2020). “Neural Predictor for Neural Architecture Search”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

White, C., W. Neiswanger, S. Nolen, and Y. Savani (2020). “A Study on Encodings for Neural Architecture Search”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

White, C., W. Neiswanger, and Y. Savani (2021a). “BANANAS: Bayesian Optimization with Neural Architectures for Neural Architecture Search”. In: *Proc. of the Conference of Artificial Intelligence (AAAI)*.

White, C., S. Nolen, and Y. Savani (2021b). “Exploring the loss landscape in neural architecture search”. In: *Proc. of the Conference on Uncertainty in Artificial Intelligence (UAI)*.

White, C., M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter (2023). “Neural Architecture Search: Insights from 1000 Papers”. In: *arXiv.org* abs/2301.08727.

White, C., A. Zela, B. Ru, Y. Liu, and F. Hutter (2021c). “How Powerful are Performance Predictors in Neural Architecture Search?” In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Williams, R. J. and D. Zipser (1989). “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks”. In: *Neural Computation* 1.2, pp. 270–280.

Wu, B., X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer (2019). “FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Wu, J., X. Dai, D. Chen, Y. Chen, M. Liu, Y. Yu, Z. Wang, Z. Liu, M. Chen, and L. Yuan (2021a). “Stronger NAS with Weaker Predictors”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Wu, Y., A. Liu, Z. Huang, S. Zhang, and L. V. Gool (2021b). “Neural Architecture Search as Sparse Supernet”. In: *Proc. of the Conference of Artificial Intelligence (AAAI)*.

Wu, Z., S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu (2021c). “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1, pp. 4–24.

Xie, C., M. Tan, B. Gong, A. L. Yuille, and Q. V. Le (2020). “Smooth Adversarial Training”. In: *arXiv.org* abs/2006.14536.

Xie, C., Y. Wu, L. van der Maaten, A. L. Yuille, and K. He (2019a). “Feature Denoising for Improving Adversarial Robustness”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xie, S., H. Zheng, C. Liu, and L. Lin (2019b). “SNAS: stochastic neural architecture search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Xilinx Inc. Vivado High-Level Synthesis (). Accessed: 2021-11-17.

Xilinx zynq-7000 soc zc706 evaluation kit (). Accessed: 2021-11-17.

Xu, D., Y. Zhu, C. B. Choy, and L. Fei-Fei (2017). “Scene graph generation by iterative message passing”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xu, K., W. Hu, J. Leskovec, and S. Jegelka (2019). “How Powerful are Graph Neural Networks?” In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Xu, K., C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka (2018). “Representation Learning on Graphs with Jumping Knowledge Networks”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Xu, Y., L. Xie, X. Zhang, X. Chen, G. Qi, Q. Tian, and H. Xiong (2020). “PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Yan, S., C. White, Y. Savani, and F. Hutter (2021). “NAS-Bench-x11 and the Power of Learning Curves”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Yan, S., Y. Zheng, W. Ao, X. Zeng, and M. Zhang (2020). “Does Unsupervised Architecture Representation Learning Help Neural Architecture Search?” In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Yang, A., P. M. Esperança, and F. M. Carlucci (2020). “NAS evaluation is frustratingly hard”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Yarkony, J. (2015). “Next Generation Multicuts for Semi-Planar Graphs”. In: *Advances in Neural Information Processing Systems (NeurIPS), Optimization in Machine Learning Workshop*.

Yarkony, J., T. Beier, P. Baldi, and F. A. Hamprecht (2014). “Parallel multicut segmentation via dual decomposition”. In: *Proc. of New Frontiers in Mining Complex Patterns Workshop*.

Yarkony, J. and C. Fowlkes (2015). “Planar Ultrametrics for Image Segmentation”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Yarkony, J., A. Ihler, and C. Fowlkes (2012). “Fast Planar Correlation Clustering for Image Segmentation”. In: *Proc. of the European Conference on Computer Vision (ECCV)*.

Yarkony, J. and S. Wang (2018). “Accelerating Message Passing for MAP with Benders Decomposition”. In: *arXiv.org* abs/1805.04958.

Yi, L., H. Su, X. Guo, and L. J. Guibas (2017). “Syncspeccnn: Synchronized spectral cnn for 3d shape segmentation”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ying, C., A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter (2019). “NAS-Bench-101: Towards Reproducible Neural Architecture Search”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Ying, Z., J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec (2018). “Hierarchical Graph Representation Learning with Differentiable Pooling”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

You, J., R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec (2018). “GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Zela, A., T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter (2020a). “Understanding and Robustifying Differentiable Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Zela, A., A. Klein, S. Falkner, and F. Hutter (2018). “Towards Automated Deep Learning: Efficient Joint Neural Architecture and Hyperparameter Search”. In: *Proc. of the International Conference on Machine Learning (ICML), AutoML Workshop*.

Zela, A., J. Siems, and F. Hutter (2020b). “NAS-Bench-1Shot1: Benchmarking and Dissecting One-Shot Neural Architecture Search”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Zela, A., J. N. Siems, L. Zimmer, J. Lukasik, M. Keuper, and F. Hutter (2022). “Surrogate NAS Benchmarks: Going Beyond the Limited Search Spaces of Tabular NAS Benchmarks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Zhang, C., F. Huber, M. Knop, and F. Hamprecht (2014). “Yeast Cell Detection and Segmentation in Bright Field Microscopy”. In: *IEEE International Symposium on Biomedical Imaging (ISBI)*.

Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang (2017). “Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising”. In: *IEEE Transactions on Image Processing* 26.7, pp. 3142–3155.

Zhang, M., S. W. Su, S. Pan, X. Chang, M. E. Abbasnejad, and R. Haffari (2021). “iDARTS: Differentiable Architecture Search with Stochastic Implicit Gradients”. In: *Proc. of the International Conference on Machine Learning (ICML)*.

Zhang, M., S. Jiang, Z. Cui, R. Garnett, and Y. Chen (2019). “D-VAE: A Variational Autoencoder for Directed Acyclic Graphs”. In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Zhao, P., P. Chen, P. Das, K. N. Ramamurthy, and X. Lin (2020). “Bridging Mode Connectivity in Loss Landscapes and Adversarial Robustness”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Zhou, J., G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun (2020). “Graph neural networks: A review of methods and applications”. In: *AI Open* 1, pp. 57–81.

Zoph, B. and Q. V. Le (2017). “Neural Architecture Search with Reinforcement Learning”. In: *Proc. of the International Conference on Learning Representations (ICLR)*.

Zoph, B., V. Vasudevan, J. Shlens, and Q. V. Le (2018). “Learning transferable architectures for scalable image recognition”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.