HELLENIC REPUBLIC
UNIVERSITY OF MACEDONIA

ΕΛ.Ι.Δ.Ε.Κ.
Ελληνικό Ίδρυμα Έρευνας & Καινοτομίας

Parallel and Distributed Methods for Autonomous Design of Artificial Neural Networks

PhD Thesis

George Kyriakides

ge.kyriakides@uom.edu.gr

Department of Applied Informatics
University of Macedonia

Supervisor:
Prof. Dr. Konstantinos Margaritis

July 5, 2022

To my family and my friends

Acknowledgements

During my academic journey, from the beginning of my Bachelor's up to the present day, Prof. Konstantinos Margaritis has greatly supported and inspired me, directly and indirectly, either through personal communication or due to their passion for his teaching subjects. Furthermore, his guidance and understanding throughout my Ph.D. have been a determining factor in completing this thesis. As such, I would like to greatly thank him for his support and faith in me throughout these years, especially during the difficult times of the pandemic. Furthermore, I would like to thank the members of the doctoral committee. Prof. Ioannis Refanidis, who was the first to introduce me to the concepts of neural networks and artificial intelligence, inspiring me to further deepen the subjects, while also providing me with various opportunities to gain academic experience throughout my Ph.D. Moreover, I would like to thank Prof. Nikolaos Samaras for introducing me to optimization and algorithmic analysis subjects during my studies and providing timely feedback early on during my Ph.D. In essence, subjects taught to me by all three committee members provided the basis upon which this thesis is built and I am grateful to have had them as my tutors and committee. Finally, I would like to thank Prof. Lazaros Iliadis for his support and trust during the AIAI/EANN conferences, as both presenting and reviewing for those conferences has provided much needed (and proved an enjoyable) experience.

Finally, I would like to thank the Hellenic Foundation for Research and Innovation which provided me with the scholarship which made the completion of this thesis possible on time.

ΕΛ.Ι.Δ.Ε.Κ.
Ελληνικό Ίδρυμα Έρευνας & Καινοτομίας

The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the HFRI PhD Fellowship grant (Fellowship Number: 646).

Abstract

In recent years, neural networks in the form of deep learning have re-ignited a widespread interest in artificial intelligence. The applications that leverage deep learning have automated several conceptionally easy tasks that are hard to express algorithmically. Examples include identifying objects in an image, counting the number of occurrences for the same object, autonomous driving and accurate speech-to-text.

While deep learning enables the automation of such tasks without explicitly instructing the machine, intricate neural architectures are employed. These architectures require several person-hours and computational resources to design them. Furthermore, expertise both in neural networks and in the specific field of application are pre-requisites.

Neural architecture search aims to automate the process of designing the networks by several different approaches, including metaheuristics, reinforcement learning, and differentiable methods. This thesis employs and studies parallel and distributed neural architecture search algorithms. The efficiency and effectiveness of the implemented algorithms are studied and speed-up techniques that are widely adopted but have not been studied in-depth before. Furthermore, under the scope of the thesis, an open-source library aimed at distributed neural architecture search has been developed. Finally, the thesis contributes to the field of neural architecture search by verifying the effectiveness and showing the limitations of methods such as reduced epoch training, distributed evolutionary search methods, distributed differentiable methods, and employing neural architecture search methods for graph convolutional networks.

Contents

Acknowledgements ii

Abstract iv

1 Introduction 1

1.1 Neural Networks 2

1.2 Accelerating Neural Architecture Search 3

1.3 Message Passing Interfaces 3

1.4 Thesis Structure and Related Publications 4

Chapter 1 References 7

2 Background on Neural Architecture Search 11

2.1 Search Spaces 11

2.1.1 Global Search Spaces 12

2.1.2 Cell Search Spaces 12

2.1.3 Hierarchical Search Spaces 15

2.2 Optimization Methods 15

2.2.1 Evolutionary Approaches 16

2.2.2 Reinforcement Learning 21

2.2.3 Gradient Based Methods 26

2.3 Evaluation Methods 30

2.3.1 Candidate Evaluation 31

2.3.2 Method Comparison 32

Chapter 2 References 35

3 A Distributed Neural Architecture Search Framework 41

3.1 The complexity, Cost, and Speedup of NAS Pipelines 41

3.2 Need for a Framework 43

3.2.1 NORD Components 45

3.3 NORD NAS Examples 58

3.3.1 A simple evolutionary algorithm 58

3.3.2 A simple reinforcement learning algorithm 62

Chapter 3 References 70

4 Relative Ranking Retention Between Optimizers 72

4.1 Introduction 72

4.2 Methodology 73

4.2.1 Evolving Architectures 73

4.2.2 Experimental Setup 74

4.3 Results 77

4.3.1 Neural Architectures 77

4.3.2 Monte Carlo Simulations 79

4.4 Limitations 81

4.5 Summary 82

Chapter 4 References 83

5 Training Candidates on Proxy Training Epochs 84

5.1 Introduction and Relevant Works 84

5.2 Methodology . . . 86
5.2.1 Generating Architectures . . . 86
5.2.2 NASBench-101 Simulations . . . 90
5.2.3 Monte-Carlo Simulations . . . 91
5.3 Results . . . 92
5.3.1 Generated Architectures . . . 93
5.3.2 NASBench-101 Results . . . 95
5.3.3 Monte-Carlo Results . . . 101
5.4 Limitations . . . 103
5.5 Summary and Conclusions . . . 103

Chapter 5 References **106**

6 Distributed Regularized Evolution for Global Search Spaces **108**
6.1 Introduction . . . 108
6.2 Related Work . . . 109
6.2.1 DENSER . . . 109
6.2.2 Auto-KERAS . . . 110
6.2.3 DeepSwarm . . . 110
6.2.4 NASH . . . 110
6.2.5 Fashion-MNIST . . . 111
6.3 Methodology . . . 111
6.3.1 Architecture Representation . . . 111
6.3.2 Evolving Architectures . . . 113
6.3.3 Implementing the Networks . . . 116
6.4 Experiments . . . 116
6.4.1 Experimental Setup . . . 116

6.4.2 Results 117
6.4.3 Discussion 120
6.4.4 Limitations 121
6.5 Summary 121

Chapter 6 References 123

7 Graph Neural Networks and Neural Architecture Search 125
7.1 Introduction 125
7.2 Related Works 126
7.2.1 Predicting neural network performance 126
7.2.2 Neural architecture search on graph neural networks 127
7.3 Methodology 128
7.3.1 GCNs and the SAGE framework 129
7.3.2 Genome representation 129
7.3.3 Graph ranking implementation 132
7.3.4 Generating the datasets 133
7.4 Experiments 133
7.4.1 Experimental setup 133
7.4.2 Experimental results 134
7.4.3 Comparing to ReNAS 139
7.5 Limitations 142
7.6 Conclusions 142

Chapter 7 References 144

8 Parallelizing Differentiable Neural Architecture Search 147
8.1 Introduction 147

8.2 Methodology and Experimental Setup 148

8.3 Results 151

8.3.1 Search Phase 151

8.3.2 Best Model Analysis 154

8.4 Limitations 156

8.5 Discussion and Future Work 156

Chapter 8 References 158

9 Conclusions and Future Directions 160

9.1 Conclusions 160

9.2 Future Directions 162

List of Figures

1.1 NAS papers per year. 2

2.1 Comparison between VGG16 architecture, global, and cell search spaces. 14

2.2 Example hierarchical search space. 16

2.3 Example evolutionary procedure under NAS. 17

2.4 Example genome encoding and the corresponding network. 18

2.5 Examples of regularized vs classic evolution for NAS. The current generation is depicted on the X-axis, while current offspring's network test accuracy is depicted on the Y-axis. 22

2.6 The controller from [1] generating a network. 24

2.7 Generating architectures, calculating training data, and sharing parameter updates. 25

2.8 NASNet-A cell architectures, as depicted in the original paper[8]. 26

2.9 Comparison of two computationally equivalent computational graphs. The one on the left utilizes edge operations (similar to DARTS), while the one on the right utilizes node operations (as in previously discussed methods). 28

2.10 The process of searching for a cell with DARTS; (from left to right) a. starting with all possible connections, b. initializing weights a, c. optimizing weights (bold edges have greater weights), d. retaining only the strongest edges for each node pair. 30

2.11 Neural architecture structure from [43]; (starting from left, clockwise) a. the outer skeleton, b. layer selection for a 7-node cell, c. lower-level calculation in the same cell. 34

3.1 Comparison of run times (search algorithm only) and network validation accuracy between a reinforcement learning and an evolutionary algorithm. 42

3.2 NAS pipeline, parallelizable components in green. Blue components have negligible relative computational cost, while red components do not contribute to overall cost or cannot be parallelized. 44

3.3 The network implemented in 3.1. 46

3.4 Logic and communication under the Environment context manager. 53

3.5 Performance of regularized evolution on NASBench-101 and NASBench-201. 62

3.6 Network validation accuracy for 500 episodes of the reinforcement learning algorithm. 69

4.1 Sample architectures evolved. 75

4.2 Genome example. 75

4.3 Comparison of achieved accuracy in 1 epoch of training to 50 epochs. 77

4.4 Rank correlation coefficients heatmap. 78

4.5 Rank correlation coefficients heatmap for the second half of generated architectures. 79

4.6 2-D Rastrigin functions, original(left) and perturbed(right) with $\tau = 0.738$ 80

5.1 DeepNEAT cell architecture. 88

5.2 Example of neural architectures generated with the mutation operator. Light cyan nodes (Sample 5) are inactive. 89

5.3 NASBench-101 fixed outer skeleton. 91

5.4 Rastrigin function 2D, 100 x 100 mesh grid. Lighter shades indicate higher values. 92

5.5 DeepNEAT accuracy distribution for 1, 5, 10, and 20 epochs, compared to 50 epochs. 94

5.6 Bootstrap estimated correlations between fully-trained (y-axis) and partially-trained (x-axis) versions. 95

5.7 Bootstrap estimated correlations between fully-trained (y-axis) and partially-trained (x-axis) versions for *rounded* accuracies. 96

5.8 Bootstrap estimated correlation standard error between fully-trained (y-axis) and partially-trained (x-axis) versions for *rounded* accuracies. 96

5.9 Accuracy distributions and Kendall's tau-b between PTGs and FTG for NASBench. 98

5.10 Bootstrap estimated distributions for Kendall's tau-b on the *rounded* NASBench dataset. The number in brackets denotes the standard error. 99

5.11 Bootstrap estimated distributions for Kendall's tau-b on the *rounded* NASBench dataset. The number in brackets denotes the standard error. 100

6.1 Sampled Fasion-MNIST images, one for each class. 112

6.2 Times an individual is sampled for serial (1 worker) and distributed (4 workers) version of regularized evolution. 114

6.3 Three node additions to the same genome. 115

6.4 Best architecture's performance (left) and performance empirical cumulative distributions (right). 118

6.5 Overlay of all genomes' architectures. 119

7.1 **Left:** Example of a genome with connection, layer, and hyper-parameter chromosomes. **Right:** The corresponding network. . . 131

7.2 Genome to network implementation. 132
7.3 Progression of the best found architecture's performance on NAS-Bench-101. 135
7.4 Progression of the best found architecture's performance on Fashion-MNIST 135
7.5 Performance boxplots for NAS-Bench-101, grouped by final aggregation layer. 136
7.6 Performance boxplots for Fashion-MNIST, grouped by final aggregation layer. 136
7.7 Scatterplot of kendall's tau (y-axis) vs number of parameters(x-axis) on NAS-Bench-101. Color indicates kendall's tau for the top-50% test set instances. 137
7.8 Scatterplot of kendall's tau (y-axis) vs number of parameters(x-axis) on Fashion-MNIST. Color indicates kendall's tau for the top-50% test set instances. 138
7.9 Best architecture found for the Fashion-MNIST derived dataset. . 140

8.1 Training accuracy curves for the search phase. 152
8.2 Validation accuracy curves for the search phase. 153
8.3 Graph edit distance values for the normal cells. 154
8.4 Graph edit distance values for the reduction cells. 154
8.5 Best models cell architectures. 155
8.6 Best model training fine-tune accuracy. 156
8.7 Best model validation fine-tune accuracy. 156

List of Tables

1.1 Table of related publications. 6

2.1 DeepNEAT hyperparameters, represented by an individual's genes. 19

2.2 Layer choices in [19] 20

3.1 NORD Datasets 55

4.1 Experimental Parameters. 76

4.2 Average training times (in seconds per architecture). 78

4.3 Mean of achieved solution's top percentage for the original and perturbed search space, when applied to the original function. . . 81

4.4 Standard deviation of achieved solution's top percentage for the original and perturbed search space, when applied to the original function. 81

5.1 Search phase and final training epochs of various methods. 86

5.2 DeepNEAT hyperparameters ranges used. 87

5.3 Average DeepNEAT accuracy for each optimizer for the fully and partially-trained versions. 93

5.4 Expected performance when searching in the original and perturbed search spaces. Expressed in the top-N percentage of the original search space (smaller is better). 102

6.1 Available layer choices. 113

6.2 Performance of similar algorithms 120

7.1 Available layer gene options. 130
7.2 Experimental Parameters. 134
7.3 Fine-tuning results on NAS-Bench-101. 139
7.4 Comparison with ReNAS on NAS-Bench-101. 141

8.1 Search and training parameters 151
8.2 Key performance metrics 152
8.3 Kolmogorov-Smirnov and Benjamini/Hochberg-corrected p-values 155

Listings

3.1 Descriptor example 45
3.2 Local evaluator example 48
3.3 Benchmark evaluators examples 50
3.4 Environment example 52
3.5 Custom dataset example 56
3.6 Regularized evolution for NASBench-101 60
3.7 Changes to apply regularized evolution to NASBench-201 62
3.8 Environment for NASBench-101 63
3.9 Reinforcement learning for NASBench-101 67

Chapter 1

Introduction

In the past decade, neural networks empowered by the development of General-Purpose Graphics Processing Units (GPGPUs) have propelled the automation of various conceptually easy but implementation-wise challenging tasks. For example, discerning between images of a cat and a dog [1], or a shoe and a skirt [2]. As humans, we know what differentiates a dog and a cat, but coding this intuitive knowledge is hard. Convolutional neural networks have allowed us to avoid this need by learning features of objects autonomously. Given that a neural network can identify an object of interest, it can perform simple deductions, such as counting the number of objects in an image (hence conceptually easy tasks). Following the success in easier domains, researchers have invented ways to leverage this feature learning ability of neural networks. For example, instead of recognizing images, networks can be trained to transfer painting styles from one image to another [3], scale-up images while enhancing their details (utilized extensively in gaming) [4], animate 3d characters in a realistic fashion [5], and even change seasons in a landscape image [6].

Although these applications utilize neural networks, they require specific architectures, usually accompanied by a high computational cost. Moreover, architectures can vary significantly between applications. Recently, researchers have been trying to automate neural networks further due to the trial-and-error process of creating these architectures and the computational and wall-clock time cost of experimenting. The same driving force behind the original explosion of neural networks, i.e., GPGPUs, is now enabling the automation of neural architecture construction. The field concerned with researching methods for this level of automation has adopted the name Neural Architecture Search, and related papers follow an exponential yearly increase (except for 2021) 1.1. There have been many breakthroughs and state-of-the-art resulting architectures throughout the years [7, 8, 9, 10], but NAS is a predominate niche field. This thesis aims to make NAS more accessible by proposing a NAS framework and library. Leveraging this library, we also study parallelized and accelerated NAS methods.

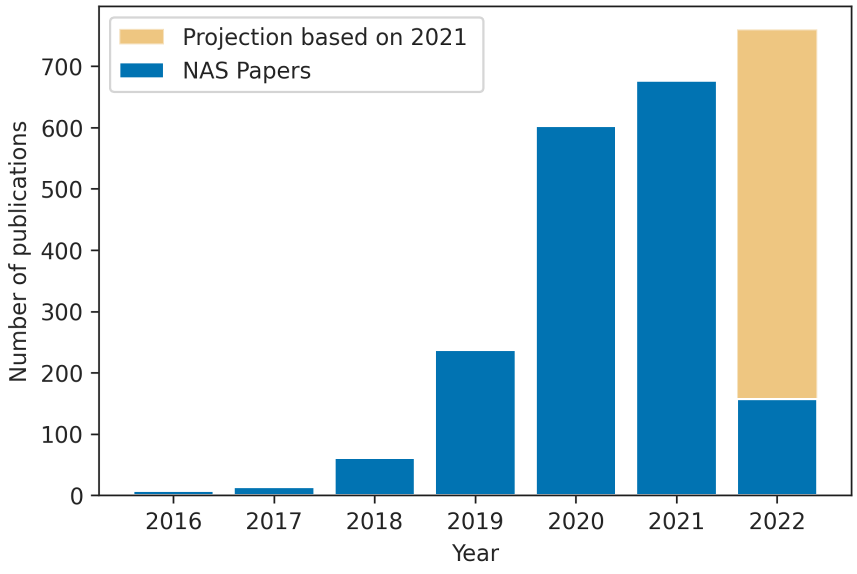


Figure 1.1: NAS papers per year.

1.1 Neural Networks

Neural networks are computational systems loosely representing a brain's structure. Their most essential component is the neuron, practically a multi-variable function. It computes a weighted sum of its inputs and then applies a non-linear (activation) function to the result. Given that an image has many pixels which are treated as individual variables, traditional neurons proved inefficient, and convolutional layers were proposed [11]. Instead of applying a specific weight to each variable, convolutional layers have a set number of filters, each with a trainable weight matrix (kernel). Kernels allow location-invariant feature learning while reducing the number of parameters required to process a specific image size. Graph Convolutional Networks [12] extend the concept of convolution to graph-like structured data, enabling feature learning from such structures.

Neural networks utilize the backpropagation of errors to optimize their weights. How these errors are utilized can significantly impact a network's training speed and quality (generalization ability). There have been proposed several optimization algorithms [13], with mini-batch gradient descent (SGD) employing Nesterov momentum and Adam being the most popular.

Training a neural network entails splitting the training data into batches (for numerical stability and memory constraints), predicting the target variable, back-propagating the errors and updating the weights accordingly. An epoch has passed when all the batches in a dataset have been processed. A network may train for a set number of epochs or until no further improvements in the validation set are observed.

1.2 Accelerating Neural Architecture Search

Due to the high computational cost of Neural Architecture Search, parallel and distributed methods were used as researchers employed a considerable number of GPU workers, even from the earliest works [7]. Other researchers attempted to lower the cost of NAS by employing proxy tasks [14, 15, 8]. Although these techniques are widely used in NAS, their impact has not been extensively studied. Using the library developed in this thesis, we study various proxy tasks and parallelization of NAS, previously unexplored or where their effects have not been previously studied.

1.3 Message Passing Interfaces

Message passing interfaces enable the communication of individual processes in distributed memory systems. MPI (Message Passing Interface) [16] is a message-passing standard designed to regularize this inter-process communication. It allows for point-to-point (i.e., from a single process to another) and collective communications (i.e., many-to-many, one-to-many, many-to-one). The most trivial functionality specified is synchronous point-to-point communication, usually in the form of send and recv (receive) functions, indicating the source/target process and data to transmit. Collective communications include:

- bcast: broadcast, sending from one process to all others.
- gather: all processes send data to a single process, usually a master.
- reduce: similar to gather but a reduction such as min/max/mean is applied to the data by the target process.
- scatter: where data from a source process is divided amongst target processes.
- barrier: a blocking operation where all processes must reach the specific code line for any operation to continue.

MPI-like collective communications have recently been incorporated in neural network libraries such as PyTorch [17] and Tensorflow [18] to allow for distributed training of neural networks. Although the work in this thesis initially utilized the Horovod library [19] for distributed training, later, native implementations of collective communications and distributed training motivated us to migrate to native PyTorch implementations.

1.4 Thesis Structure and Related Publications

This thesis, as already mentioned, proposes a NAS framework and studies parallel NAS algorithms and acceleration techniques. Given that the research was conducted during the explosion of NAS literature, we focused on the most impactful, contemporary methods and techniques. The thesis is structured in an incremental manner, where each chapter utilizes concepts and knowledge from previous chapters. Below each chapter is briefly presented, while Table 1.1 summarizes publications related to each chapter.

Chapter 2 introduces basic NAS concepts and algorithms. NAS search spaces, search methods, and evaluation methods are discussed. Finally, we present the reinforcement learning approach of the NAS paper [7]2016 that sparked extended interest in NAS, genetic and evolutionary methods, as well as the paper that introduced backpropagation as means to optimizing architectures.

Chapter 3 presents NORD, our proposed NAS library. Essential components of NORD such as descriptors, evaluators and the distributed environment are presented together with examples and implementations of basic NAS algorithms.

Chapter 4 studies the effect of various optimizers on relative ranking between trained architectures. As some optimizers tend to converge faster, we study a reduced-epochs approach to evaluating neural architectures and the effect on fully trained architectures. Here, distributed computing is utilized to evaluate the architectures.

Chapter 5 studies the effect of reduced epoch training, building on chapter 4 and extending the study to benchmark datasets, and larger networks. Te results further show that noise cannot be avoided for proxy tasks, although its merits can potentially out-weight the drawbacks of induced noise.

Chapter 6 employs a distributed method that combines elements from two NAS methods, DeepNEAT [20], and Regularized Evolution [21], and utilizes proxy training tasks to find optimal architectures in a global search space for the Fashion-MNIST dataset, achieving a state-of-the-art accuracy for global search spaces.

Chapter 7 utilizes the method from chapter 6 to find a graph-convolutional architecture that can predict the relative performance of other neural architectures. By utilizing the graph structure of the dataset architectures instead of feature maps (as was previously done), the method achieves better results than the previous state-of-the-art model.

Chapter 8 studies parallelization effects for differentiable NAS methods. As [9] provided considerable computational cost reduction, most work conducted on this sub-field of NAS is concerned with improving the original method. As such, parallelization of the method had not been previously studied.

Chapter 9 discusses limitations and future directions.

Table 1.1: Table of related publications.

Chapter	Publication	Type
2	An introduction to neural architecture search for convolutional networks [22]	arXiv pre-print
3	Towards automated neural design: An open source, distributed neural architecture research framework [23]	Conference
3	NORD: A python framework for Neural Architecture Search [24]	Journal
4	Comparison of Neural Network Optimizers for Relative Ranking Retention Between Neural Architectures [25]	Conference
5	The effect of reduced training in neural architecture search [26]	Journal
6	Regularized Evolution for Macro Neural Architecture Search [27]	Conference
7	Evolving graph convolutional networks for neural architecture search [28]	Journal
8	The Effectiveness of Synchronous Data-parallel Differentiable Architecture Search, *Approved for publication, EANN2022*	Conference

Chapter 1 References

[1] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[2] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[3] Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu, and M. Song, "Neural style transfer: A review," *IEEE transactions on visualization and computer graphics*, 2019.

[4] A. Watson, "Deep learning techniques for super-resolution in video games," *arXiv preprint arXiv:2012.09810*, 2020.

[5] F. G. Harvey, M. Yurick, D. Nowrouzezahrai, and C. Pal, "Robust motion in-betweening," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 60–1, 2020.

[6] L. Karacan, Z. Akata, A. Erdem, and E. Erdem, "Manipulating attributes of natural scenes via hallucination," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 1, pp. 1–17, 2019.

[7] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 11 2016. [Online]. Available: http://arxiv.org/abs/1611.01578

[8] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 7 2018. [Online]. Available: http://arxiv.org/abs/1707.07012

[9] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *7th International Conference on Learning Representations, ICLR 2019*, 6 2018. [Online]. Available: http://arxiv.org/abs/1806.09055

[10] M. S. Tanveer, M. U. K. Khan, and C.-M. Kyung, “Fine-tuning darts for image classification,” in *2020 25th International Conference on Pattern Recognition (ICPR).* IEEE, 2021, pp. 4789–4796.

[11] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, “Object recognition with gradient-based learning,” in *Shape, contour and grouping in computer vision.* Springer, 1999, pp. 319–345.

[12] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” pp. 1024–1034, 2017.

[13] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747*, 2016.

[14] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, “Practical block-wise neural network architecture generation,” pp. 2423–2432, 2018.

[15] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, “Hierarchical representations for efficient architecture search,” *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 11 2018. [Online]. Available: http://arxiv.org/abs/1711.00436

[16] D. W. Walker and J. J. Dongarra, “Mpi: a standard message passing interface,” *Supercomputer*, vol. 12, pp. 56–68, 1996.

[17] A. Paszke, 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, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[19] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[20] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312.

[21] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, 7 2019.

[22] G. Kyriakides and K. Margaritis, "An introduction to neural architecture search for convolutional networks," *arXiv preprint arXiv:2005.11074*, 2020.

[23] G. Kyriakides and K. Margaritis, "Towards automated neural design: An open source, distributed neural architecture research framework," in *Proceedings of the 22nd Pan-Hellenic Conference on Informatics*, 2018, pp. 113–116.

[24] G. Kyriakides and K. Margaritis, "Nord: A python framework for neural architecture search," *Software Impacts*, vol. 6, p. 100042, 2020.

[25] G. Kyriakides and K. Margaritis, *Comparison of Neural Network Optimizers for Relative Ranking Retention Between Neural Architectures*, 2019, vol. 559.

[26] G. Kyriakides and K. Margaritis, "The effect of reduced training in neural architecture search," *Neural Computing and Applications*, pp. 1–12, 2020.

[27] G. Kyriakides and K. Margaritis, “Regularized evolution for macro neural architecture search,” in *IFIP International Conference on Artificial Intelligence Applications and Innovations.* Springer, 2020, pp. 111–122.

[28] G. Kyriakides and K. Margaritis, “Evolving graph convolutional networks for neural architecture search,” *Neural Computing and Applications*, vol. 34, no. 2, pp. 899–909, 2022.

Chapter 2

Background on Neural Architecture Search

This chapter presents basic Neural Architecture Search components and algorithms. All of the material in this chapter is utilized later in the thesis. A good understanding is beneficial and allows the reader to follow the rest of the chapters more quickly.

2.1 Search Spaces

The first component of NAS which significantly impacts the generated architectures structure is the search space from which these architectures are generated. The search space directly dictates the possible architectures that can be generated and evaluated from the search algorithm. For example, assuming that a neural network can be represented by a computational graph G, the search space defines a set of all possible graphs S for the given NAS method. As such, a NAS algorithm aims to select the best computational graph $G^* \in S$. Search spaces can be distinguished into global search spaces, cell search spaces, and hierarchical search spaces. In this chapter, we present all three approaches and some notable examples.

2.1.1 Global Search Spaces

The first, more general approach is to define a global optimization search space, which allows the optimization algorithm to generate arbitrary network architectures. There are relatively few restrictions concerning the network structure in a global search space. The algorithm determines the network's layers type, hyper-parameters, and inter-layer connectivity. Usually, some restrictions exist, such as the input and output structure of the network, But the design algorithm is free to decide for the majority of the network's architecture.

A *global search space* is the first search space utilized for modern NAS in [1], consisting of successive convolutional layers of variable filter number, height, width and stride with skip connections. Many other works also utilized global search spaces following this original NAS work. In DeepNEAT [2] the authors also utilize a global search space of layer blocks, where each block utilizes a convolutional layer of variable filter number and size, followed by a dropout and possibly a pooling layer. The algorithm optimizes these variables and the connections between (but not within) blocks. Furthermore, each candidate architecture optimizes a set of training hyper-parameters (such as optimizer and data augmentation parameters). DENSER [3] creates networks at least 3 layers deep and up to a maximum of 30 convolutional and pooling layers and up to 10 fully-connected layers. In Deepswarm [4], the search space consists of sequentially connected networks consisting of convolutional, pooling and batch normalization layers.

2.1.2 Cell Search Spaces

Realizing that the global search spaces are possibly too big for design algorithms to be efficient, researchers started imposing restrictions on the size and diversity of search spaces. Furthermore, inspired by the observation that many state-of-the-art handcrafted-architectures exhibited repetitive patterns in their design, such as ResNet [5] and VGG [6], The researchers started utilizing templates with repetitive patterns and utilized NAS algorithms in order to optimize the architecture of these patterns. These patterns in NAS literature are referred to as cells, and consequently, search spaces that utilize and optimize cell architectures are called *cell search spaces*.

Employing cell search spaces limits the optimization algorithm's freedom to generate arbitrary networks but narrows the search into regions of the global search space, where suitable architectures are expected to be found. Utilizing cell search spaces requires prior knowledge regarding architectures that work well in the given application domain. Furthermore, considerably more time has to be invested in designing the search space when compared to global search spaces. Nonetheless, cell search spaces have provided state-of-the-art networks for various datasets, such as Fashion-MNIST [7].

A common cell search space is the NASNet search space, first proposed in [8]. NASNet optimizes two distinct cell types; normal and reduction cells. Normal cells are stacked on top of each other, and after every Nth normal cell, a reduction cell follows. Each cell receives as input the output of its two previous cells. As such, cell i should receive input from cells $i-1, i-2$. Inside the cell, there are B hidden states, which apply an operation (convolutional or pooling layer) to the outputs of two previous hidden states and combine the results. Many later works utilize cell search spaces. Some of them, such as [9], refrain from searching both a normal as well as a reduction cell. Instead, a max-pooling operation is applied after every Nth cell to reduce the internal representation's size. Normal cells dictate that all layers will have a stride of 1, while reduction cells dictate that their layers have a stride of 2. In [10], normal cells dictate that their layers will have a stride of 1, while reduction cells dictate that their layers have a stride of 2. This work led to [7], which by fine-tuning the macro-architecture template to include an attention module after each cell, fixed operations between specific cells and a dual stem approach, were able to generate the state-of-the-art network for Fashion-MNIST. Figure 2.1 compares a typical cell search space to a global search space and the VGG16 architecture [6]. VGG16 contains a repeating pattern of two or three convolutional layers followed by a reduction layer. A typical cell search space mimics this repeating pattern by stacking N normal cells, followed by a reduction cell.

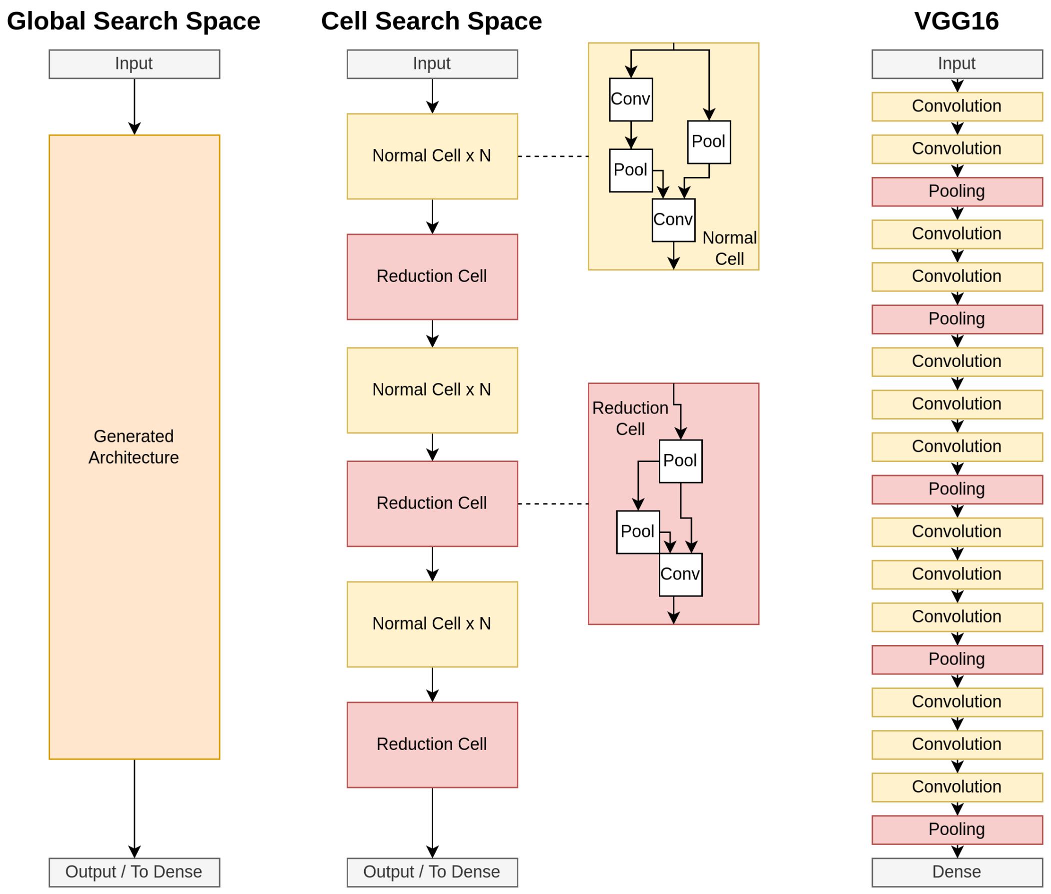


Figure 2.1: Comparison between VGG16 architecture, global, and cell search spaces.

2.1.3 Hierarchical Search Spaces

Cell search spaces provide means to leverage prior domain-specific knowledge, leading to state-of-the-art networks for given datasets. Consequently, their main disadvantage is the requirement for this prior knowledge and the possibility of introducing bias, excluding regions of the global search space where even better-performing architectures may exist. Furthermore, this prior knowledge requirement can apply even when searching for an architecture in a well-known dataset, where traditional metrics (such as top K accuracy) accompany novel ones (such as inference latency). In such cases, the existing knowledge regarding well-performing architectures may not translate well to the novel task.

A relevant case can be found in [11], where the algorithm searches for a well-performing architecture, but latency is evaluated in real-time on mobile hardware (imposing a novel metric). Here, the authors propose an intermediate solution for the search space. A hierarchical search space is proposed instead of having a fixed macro-architecture template of repeating cells. The search space consists of 7 independent blocks, organized sequentially. Each block B_i contains N_i repeating cells, where N_i is a hyper-parameter, subject to optimization.

Another hierarchical search space is proposed in [12]. In this work, N different design patterns are utilized, called motifs. Each motif of level n is a directed acyclic graph, where each of the graph's nodes is a motif of level $n-1$. Level 1 motifs are various layer types (such as convolutional and pooling layers). For example, NASNet search space in [8] can be seen as a special case of this hierarchical search space, where there are only two types of level 2 motifs and a single pre-determined level 3 motif (the macro-architecture template). Figure 2.2 depicts a four-level hierarchical search space. Finally, [2] utilizes a search space which can be seen as a special case of the one in [12] where $N = 3$.

2.2 Optimization Methods

Long before the advent of deep learning, there have been efforts to automate the design of dense network architectures through evolutionary algorithms [13]

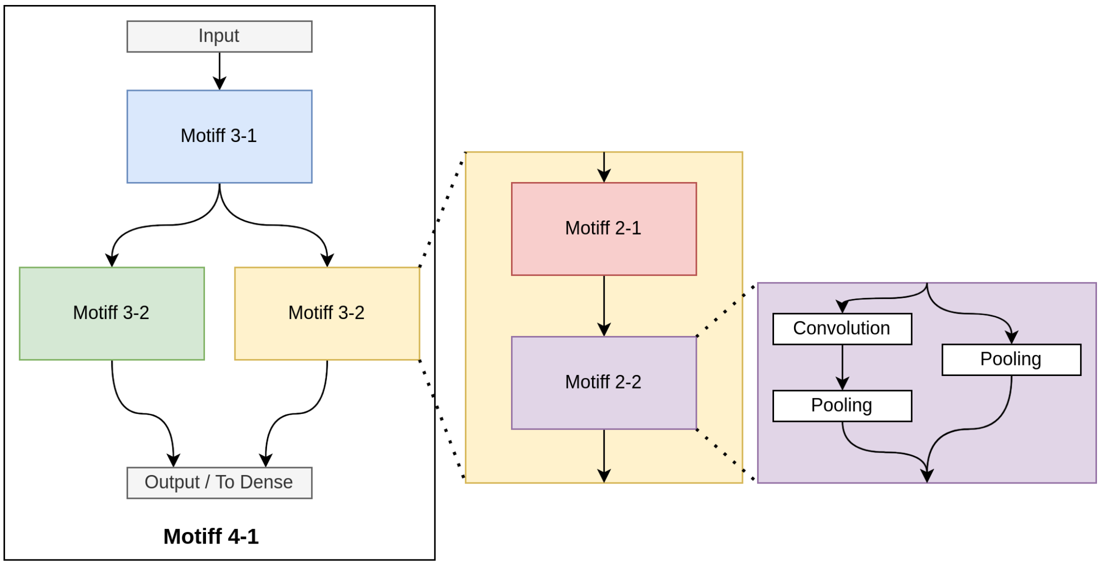


Figure 2.2: Example hierarchical search space.

while also attempting to find optimal weight values. The architecture's effect on the network's modeling capabilities was noticed, despite the simplicity of the networks, when compared to modern deep architectures. Modern NAS methods utilize several diverse optimization approaches to the problem of network architecture, while network weights are usually optimized through gradient descent. This section is concerned with the presentation of various optimization methods, mainly those utilized in later parts of the thesis, and some approaches that greatly influenced the field.

2.2.1 Evolutionary Approaches

Evolutionary algorithms (EAs) are a family of metaheuristic methods for finding solutions to non-trivial, nonlinear problems. One of the earliest works describing a computational system adapting to its environment can be found in [14]. Inspired by biological evolution, the key components of an EA are the mechanisms of selection, reproduction, crossover and mutation [15]. A population of individuals (genomes), each representing a specific solution to the same problem (where each variable is encoded as a gene), is evaluated and assigned a fitness score (i.e., how good is the proposed solution to the problem). Based on these scores and utilizing a selection strategy, the algorithm chooses a subset of the population to reproduce. Reproduction can involve crossover and mutation. Crossover is a binary operator, while mutation is a unary operator. After the reproduction process is over, the population contains new individuals who should be evaluated. This process is repeated either for a set number of iterations (called generations) or until a satisfactory solution is produced. Figure 2.3 depicts the process under a NAS framework.

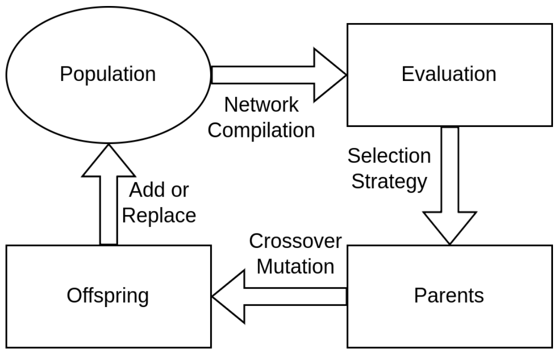


Figure 2.3: Example evolutionary procedure under NAS.

Evolutionary programming, which followed a similar path to simulating natural evolution, had been applied to the domain of neural networks by one of the domain's pioneers, Lawrence J Fogel, as early as 1990, in a paper titled "Evolving Neural Networks" [16]. Although focused on evolving neuron weights, other researchers had already experimented with generating architectures for dense networks [17]. These early works and neural network applications, in general, were limited due to the computational resources of the time. As such, researchers did not widely adopt them. More recent works, leveraging the abstraction in neural architecture that convolutional networks have provided (i.e., logically grouping neurons in layers and going beyond sequential networks), encode layer types, connectivity, and hyperparameters in their genes. Figure 2.4 depicts a simple example of such an encoding. Their choices regarding representations greatly influence the mutation and crossover operators' design. Mutation-only methods are easier to implement as they do not require tracking of gene origins to enable crossover.

Furthermore, mutations can be implemented with function-preserving transformations, such as network morphisms, allowing the inheritance of weight values. Finally, in each generation, there are several individuals for evaluation (i.e., training and testing the corresponding network); these methods provide the most straightforward approach to parallelization. Since each network can be evaluated by a different GPU worker, with minimal communication overhead, parallelization is trivial. Only the network description must be sent to the worker while the fitness value is returned, imposing minimum communication overhead.

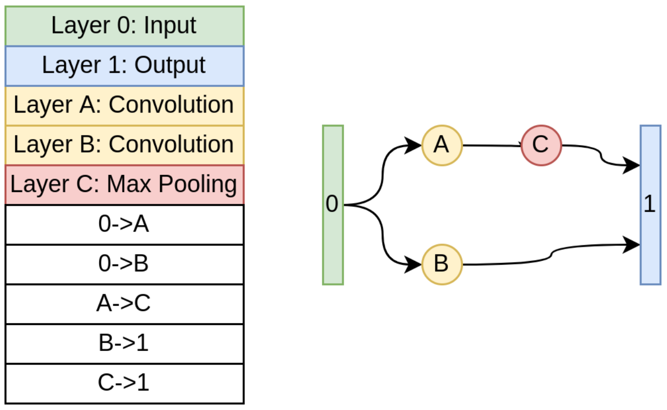


Figure 2.4: Example genome encoding and the corresponding network.

DeepNEAT and CoDeepNEAT

DeepNeat is one of the earlier works utilizing genetic algorithms for the evolution of convolutional architectures [2]. Both mutations, as well as a crossover, are employed, following earlier work conducted on evolving topologies for dense networks [13]. The method utilizes a global search space, where each network consists of arbitrarily connected convolutional blocks. These blocks consist of the following layer sequence; a 2d convolutional layer, a dropout layer and a max-pooling layer. Each block can be parameterized as follows: the convolutional layer has a set number of filters in the range of [32, 256], a kernel size of either 1 or 3, and its initial weights are scaled by a factor in the continuous range of [0, 2]. The dropout probability is in the range of [0, 0.7], while there is an option to omit the max-pooling layer. Furthermore, global hyperparameters are also optimized.

Table 2.1: DeepNEAT hyperparameters, represented by an individual's genes.

	Per-Block Hyperparameter	Range
Convolution Layer	Number of Filters	[32, 256]
	Initial Weight Scaling	[0, 2.0]
	Kernel Size	{1, 3}
Dropout Layer	Dropout Rate	[0, 0.7]
Pooling Layer	Max Pooling	{True, False}
	Global Hyperparameter	**Range**
Optimizer	Learning Rate	[0.0001, 0.1]
	Momentum	[0.68, 0.99]
	Nesterov Accelerated Gradient	{True, False}
Data Augmentation	Hue Shift	[0, 45]
	Saturation/Value Shift	[0, 0.5]
	Saturation/Value Scale	[0, 0.5]
	Cropped Image Size	[26, 32]
	Spatial Scaling	[0, 0.3]
	Random Horizontal Flips	{True, False}
	Variance Normalization	{True, False}

These include optimizer parameters, such as learning rate and momentum values, Nesterov's momentum utilization, and data augmentation parameters. Table 2.1 depicts the hyperparameters represented by each individual's genes. In order to use a crossover operator, the researchers track the origins of each gene, which helps to identify homologous genes between two parents. This tracking, in turn, allows the algorithm to decide how to treat homologous and non-homologous genes. Mutation involves the perturbation of selected values.

CoDeepNEAT extends DeepNEAT into a hierarchical search space. Here, the search space consists of two levels; the first is the design of small network patterns utilizing DeepNEAT. Then, blueprints are generated, which create networks using the patterns generated in the first level. Here, two population are co-evolved [18]. One population concerns the generation of the small patterns, while the other concerns the generation of blueprints. An individual's fitness depends on other individuals. For example, a blueprint individual's fitness depends both on the blueprint architecture (how the patterns are connected) and the architecture of each pattern. Similarly, a pattern individual may contribute to a blueprint's

Table 2.2: Layer choices in [19]

identity	1x3 then 3x1 convolution
3x3 average pooling	1x7 then 7x1 convolution
3x3 max pooling	3x3 dilated convolution
5x5 max pooling	1x1 convolution
7x7 max pooling	3x3 convolution
5x5 depthwise-seperable conv	3x3 depthwise-separable conv
7x7 depthwise-separable conv	

performance but is not solely responsible for it. The authors choose to assign a blueprint individual the fitness of its generated network, while pattern individuals are assigned the average fitness of the five best blueprints they participated in.

Regularized Evolution

In [19], the authors utilize an evolutionary approach for cell search spaces. Specifically, they utilize the NASNet search space [8], where reduction cells are added after N normal cells. Given that the search space is very well defined, each individual's genes dictate the layers and connectivity of a normal and a reduction cell. Possible layer choices (operations or "ops") include several convolution and pooling setups,as well as an identity layer, summarized in table 2.2.

The proposed algorithm retains a population of P individuals and selects S random samples as candidate parents at each generation, repeated for G. The one with the highest fitness score is selected to reproduce from these candidates. Offspring is generated by applying a mutation operator and evaluating the resulting network. The oldest individual is discarded when the offspring is inserted back into the population. The authors argue that this acts as regularization in the evolutionary process, as individuals with a high fitness score attributed to favorable stochastic conditions (such as network weight initialization, training trajectory and others) will not linger in the population. As such, after P generations, the population is entirely new, and only genes from individuals with sufficiently well-performing architectures have a high enough probability of surviving through their offspring.

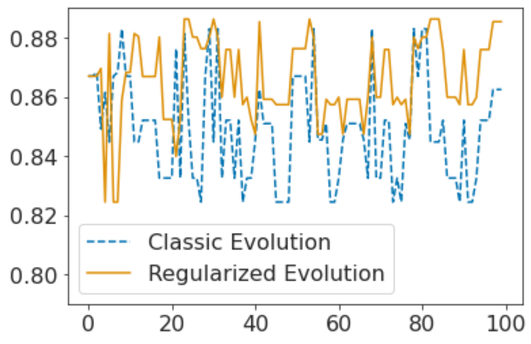
Figure 2.5a depicts a small comparison example between regularized and classic evolution. Small networks consisting of five sequential layers are trained for 108 epochs on the CIFAR-10 dataset [20]. Each layer is either a 1x1 convolution, a 3x3 convolution, or a 3x3 max-pooling layer. Mutation consists of selecting a single layer and randomly assigning a new one with a probability of 0.33 for each selection, while validation accuracy is used as the fitness function. The algorithm runs with $P = 50, S = 50$ (version 1) for 100 generations. The X-axis depicts the generation number, while the y-axis depicts the current offspring's network test accuracy. Figure 2.5b depicts the full run of 1000 generations.

As it is evident by the graphs, the regularization helps to filter out lucky evaluations, as regularized evolution consistently outperforms classic evolution. Nonetheless, the algorithm's hyper-parameters influence the regularization effect's magnitude. Repeating the experiment with $P = 50, S = 25$ (version 2, Figures 2.5c, 2.5d), we observe a less profound but still visible effect of regularization. For $P = 25, S = 10$ (version 3, Figures 2.5e, 2.5f) we observe that classic evolution outperforms its regularized version for some specific epochs.

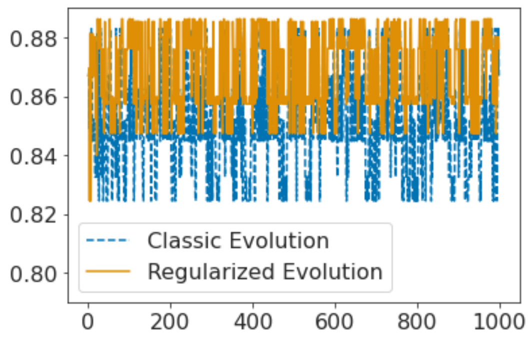
2.2.2 Reinforcement Learning

Reinforcement learning is a sub-field of machine learning, which significantly differs from supervised learning. Under a reinforcement learning scheme, instead of learning how to map input variables to a target variable, the model learns how to map its inputs to optimal actions. An agent usually utilizes the model to act optimally within a specific environment. Depending on the problem at hand, the agent may be rewarded after a single action or multiple successive actions by reaching a specific point in the environment. Reinforcement learning problems are usually formulated as a Markov Decision Process (MDP), which is defined by a 5-tuple (S, A, P, R, γ):

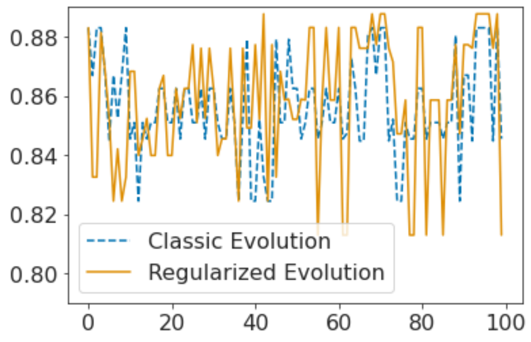
- The set of all available states S in the environment.
- The set of all available actions A to the agent.



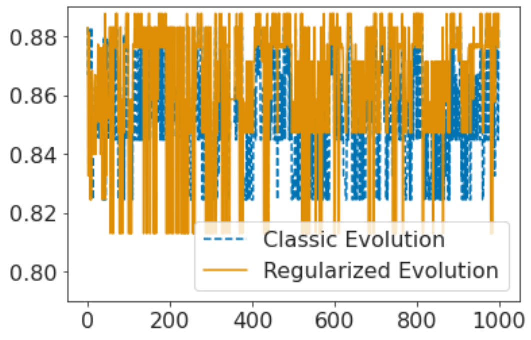
(a) $P = 50, S = 50, G = 100$



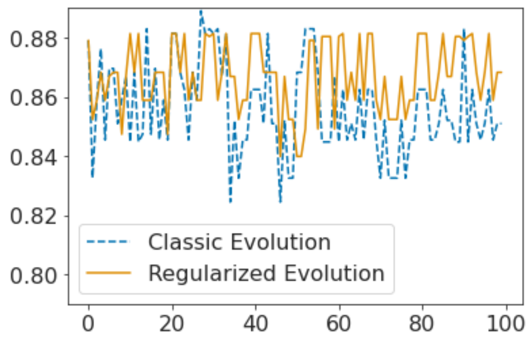
(b) $P = 50, S = 50, G = 1000$



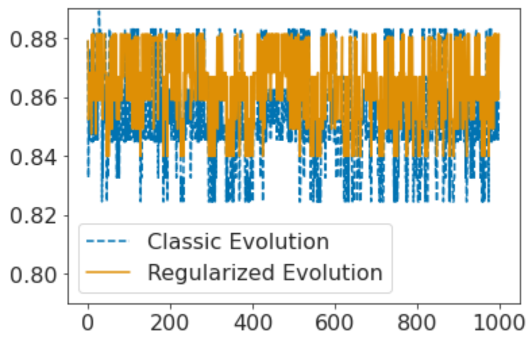
(c) $P = 50, S = 25, G = 100$



(d) $P = 50, S = 25, G = 1000$



(e) $P = 25, S = 10, G = 100$



(f) $P = 25, S = 10, G = 1000$

Figure 2.5: Examples of regularized vs classic evolution for NAS. The current generation is depicted on the X-axis, while current offspring's network test accuracy is depicted on the Y-axis.

- The transition probabilities P, where $P_a(s, s')$ is the probability of transitioning from state s to s', given that action a was performed.

- The reward function R, which defines the agent's reward $R_a(s, s')$ given that it transitioned from state s to state s' by performing action a.

- The discount factor $\gamma \in [0, 1]$, which regulates the importance between immediate and future rewards.

In a typical reinforcement learning setup, the agent can observe only part of its environment, given its current state. Thus, to act optimally, the agent must first explore its environment and gather relevant data concerning the possible states, actions, and rewards. After the agent has sufficient information, it can decide on an optimal policy π^*, which dictates its behavior. Assuming a discrete-time MDP, the agent interacts with the environment at discrete time steps. At each time step t, being in a state $s_t \in S$, the agent selects an action $a_t \in A$ and at time step $t+1$ transitions to state $s_{t+1} \in S$, while receiving reward r_{t+1}. When a terminal state is reached at time T, discounted cumulative rewards G_t can be calculated for each time step t as follows:

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+1+k} \tag{2.1}$$

The agent's trajectory (often called an episode) is used to train the model. The relevant states, actions, and discounted cumulative rewards are utilized for each time step to train a model. Depending on the reinforcement learning algorithm, how this information is utilized can vary. For example, Q-learning [21] learns the value of each possible action at each possible state by retaining a table. Deep Q-learning replaces the table with a deep neural network [22]. Advantage Actor-Critic [23] utilizes a deep neural network with two output layers; one layer models the expected value of the current state (critic) while the other models the expected value of each possible action (actor). As backpropagation of errors occurs for the actor layer only locally (errors do not propagate to the main body of the neural network), the network can learn an internal representation of the environment from the critic. In contrast, the actor only leverages this internal representation and does not contribute to its training.

Neural Architecture Search with Reinforcement Learning

One of the most influential papers in the field of NAS, and the one that coined the acronym, utilized reinforcement learning in order to design networks in a global search space [1] consisting of sequential layers with a skip-connection. Representing a neural architecture in a machine learning-friendly data structure is the first obstacle when considering reinforcement learning for NAS. In this paper, the authors represent a network's architecture as a variable-length string and utilize a Recurrent Neural Network (RNN) controller to generate the string. Reinforcement learning is used in order to train the controller, using the REINFORCE algorithm [24], by utilizing the validation accuracy of the final network as the reward. For each layer, the RNN controller selects the filter height and width, stride height and width, the number of filters, and previous layers as skip-connect inputs for the layer. The controller and architecture generation procedure are depicted in Figure 2.6.

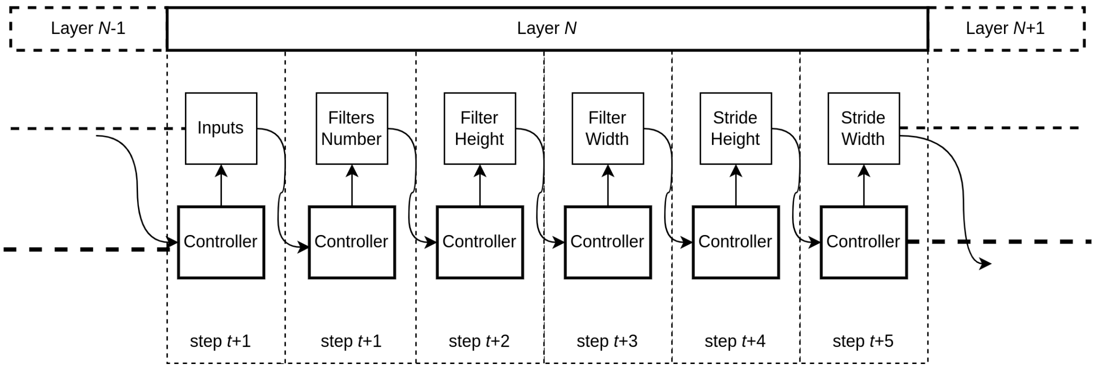


Figure 2.6: The controller from [1] generating a network.

As reinforcement learning requires a considerable amount of sampled data to generate a successful policy and episodes in this method are expensive (they require fully training and testing a network on the target dataset), the authors employ a distributed training approach. A number of S parameter-server shards (copies) is generated, where the parameters of K controller replicas are retained. Each controller replica samples m unique architectures and trains them for a set number of epochs. Each controller calculates the gradients related to the networks it samples and sends them to the parameter server to update all controller replicas' weights. Figure 2.7 summarizes the procedure, which can be decomposed to the following steps:

- For each controller, generate m architecture strings.
- For each string, compile the corresponding network.
- Train the network for e epochs and save the validation accuracy a.
- Calculate the return of the terminal state by subtracting a baseline b from a.
- Calculate the discounted rewards for all state-action pairs in the trajectory.
- Calculate the gradients θ for each controller using the mini-batch of m architectures.
- Send θ from the controller replica to the parameter servers.
- Publish θ from the parameter servers to the other replicas.

2.7.

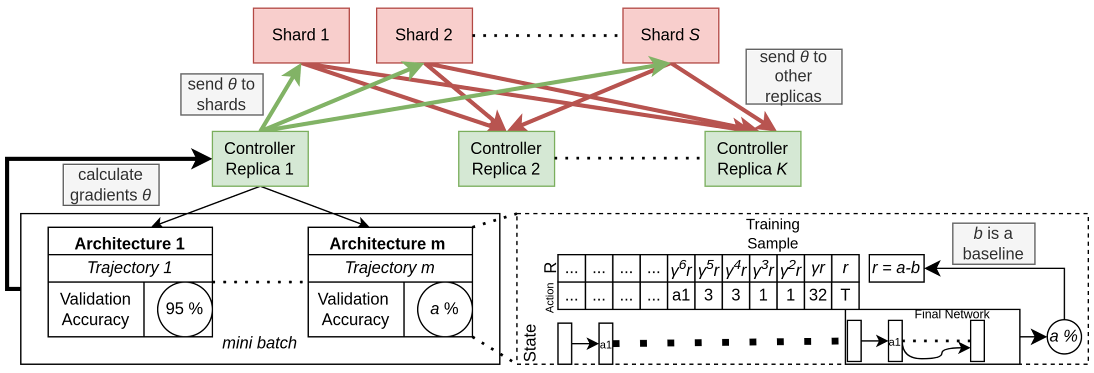


Figure 2.7: Generating architectures, calculating training data, and sharing parameter updates.

Although reinforcement learning is generally less efficient than other methods, requiring more GPU hours per generated final architecture [25], it has played a significant role in pioneering the field. The original NASNet paper [8] utilized reinforcement learning to generate cells for its novel cell search space, where each cell consists of B internal hidden states or blocks. Here, the RNN controller selected for each block inside the cell the following parameters; two previous hidden states as inputs, an operation (for example, a convolution or pooling layer) to apply to each input, as well as a method to combine the operations' results (such as addition or concatenation). The resulting network architecture with 5 blocks per cell, NASNet-A (Figure 2.8), outperformed various human-designed architectures on the ImageNet dataset [26], such as Inception V2 [27], Inception V3 [28], Xception [29], and Inception ResNet V2 [30], requiring less parameters while also providing better Top-1 and Top-5 accuracy.

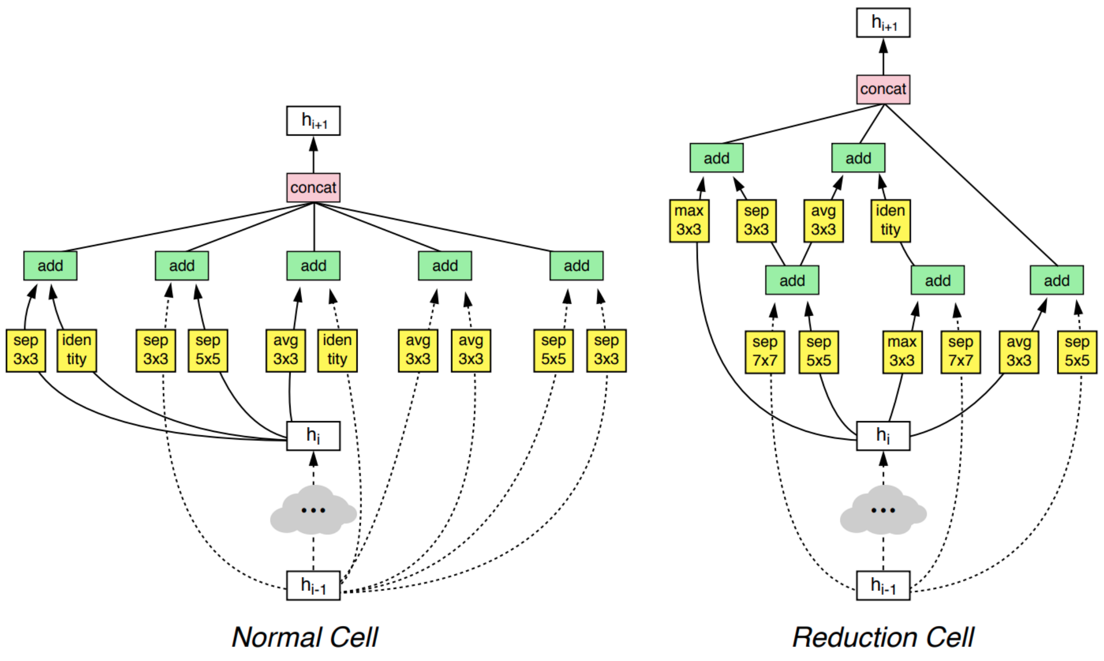


Figure 2.8: NASNet-A cell architectures, as depicted in the original paper [8].

2.2.3 Gradient Based Methods

Although evolutionary and reinforcement learning methods could provide state-of-the-art architectures, they were significantly inefficient. This inefficiency can largely be attributed to two main factors. The first is the need to train each architecture from scratch, even if it is a slight variation of a previously examined architecture. The second is the discrete representation of architectures in both approaches. These factors significantly increase the computational resources required to generate networks. Gradient-based methods were consequently developed to overcome these limitations.

DARTS: Differentiable Architecture Search

Differentiable Architecture Search (DARTS) [10] proposed a novel way of generating as well as representing neural architectures. The methods discussed previously represented neural architectures as graphs, with nodes as computational elements and edges as data-flow indicators. DARTS also uses computational graphs, with edges representing both computations and data flow. Nodes only represent the concatenation of results generated by the computations occurring in the incoming edges. Leveraging this representation, DARTS relaxes the discrete nature of neural architectures (i.e., having a specific layer type) by allowing O edges between nodes, one for each available operation. Edges are weighted, and for each node, all operations with the same origin node apply a softmax operation. A comparison between the two representations for a computationally equivalent network can be seen in Figure 2.9. Notice that the edge-operations representation (left) is more compact, requiring five nodes and six edges, while the node-operations representation (right) requires ten nodes and eleven edges.

We can consider a set $O = \{o_1, o_2, ..., o_n\}$ of weighted, same-origin operations as a mixed-operation layer (MixOp layer) $\bar{o}$, with weights a. As such, by applying a softmax over all operations' weights, the final output of the MixOp can be calculated by equation 2.2, assuming that the output of an operation o is $o(x)$.

$$\bar{o}(x) = \sum_{o \in O} \frac{e^{a_o}}{\sum_{o' \in O} e^{a_{o'}}} o(x) \tag{2.2}$$

Darts utilizes MixOp layers during the search, which enables the backpropagation of errors for both layer weights w and architecture weights a. A super network (supernet) is generated, where all possible connections and operations

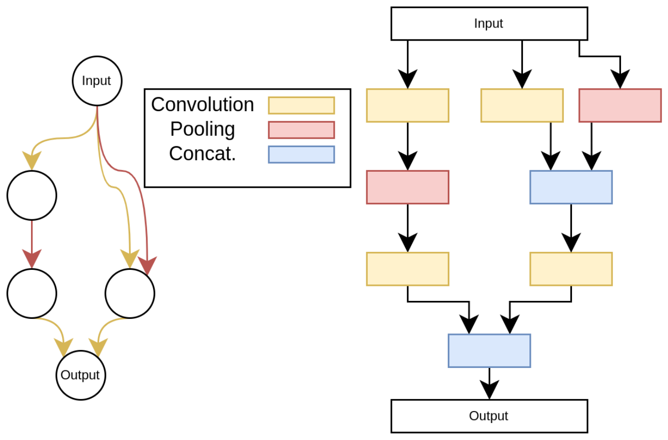


Figure 2.9: Comparison of two computationally equivalent computational graphs. The one on the left utilizes edge operations (similar to DARTS), while the one on the right utilizes node operations (as in previously discussed methods).

exist simultaneously. Cells of the same type share architecture weights, with normal cells having a stride of 1 for their layers while reduction cells have a stride of 2. In each cell, nodes at level i are connected to all lower-level nodes within the cell. DARTS employs identity operations, which enable skip-connections to exist between nodes and "Zeroize" operations, which effectively disable the connection between nodes. The set of all operations O employed by DARTS are the following:

- Zeroize
- Identity
- 3x3 max pooling
- 3x3 average pooling
- 3x3 separable convolution
- 5x5 separable convolution
- 3x3 dilated separable convolution
- 5x5 dilated separable convolution

In order to optimize both w and a, DARTS solves a bilevel optimization problem. At each training step, the weights of each individual layer are first optimized utilizing the training set (L_{train}) , while the architecture weights are then optimized utilizing the validation set (L_{val}) . Thus, the optimization problem is to minimize the loss on the validation set w.r.t. a (2.3a), given the layer weights $w*$, as they were optimized on the training set (2.3b).

The computational implementation involves sampling a mini-batch from the training set to calculate L_{train} and backpropagate the errors for w, given a. Then, a mini-batch from the validation set is utilized to calculate loss L_{train} and backpropagate the errors for a given the new values of w. The process repeats for a set number of epochs. After the search concludes, the operation with the strongest weight is selected, while the others are discarded to generate the final network. The final network is then trained from scratch for an arbitrary number of epochs. The process of starting from all possible connections as MixOps, optimizing the weights, and retrieving the final network is depicted in Figure 2.10 for a 4-node cell with $|O| = 2$.

$$\min_{a} \quad L_{val}(w^*(a), a) \tag{2.3a}$$

$$\text{s.t.} \quad w^*(a) = \arg\min_{w} L_{train}(w, a) \tag{2.3b}$$

Although DARTS was able to generate comparable but not vastly superior architectures to previous works such as [8], it required considerably less computational power to do so. On the CIFAR-10 dataset [20], DARTS needed 4 GPU-days to generate networks with 2.76 ± 0.09 Top-1 test error, while [8] needed 2000 GPU-days to generate networks with comparable performance. Other approaches need from 300 [12] to over 3000 [19] in order to generate comparable or worse networks. The main drawback of DARTS is its inability to be applied to global search spaces, mainly due to computational constraints. In the original paper, the final network contains approximately 3.3 million parameters which require approximately 12.6 MB of RAM.

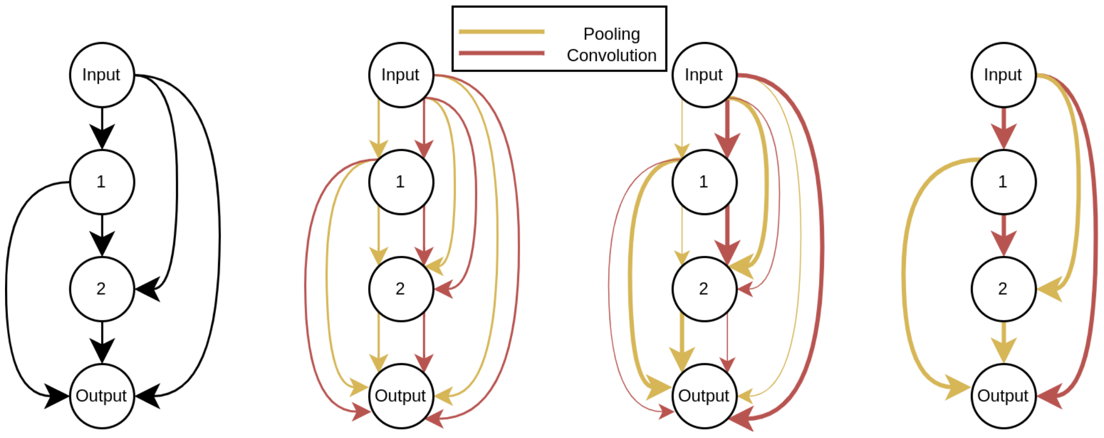


Figure 2.10: The process of searching for a cell with DARTS; (from left to right) a. starting with all possible connections, b. initializing weights a, c. optimizing weights (bold edges have greater weights), d. retaining only the strongest edges for each node pair.

Nonetheless, during the search phase, up to 10 GB of GPU RAM are utilized due to the need to store gradients and interim results for MixOps. Another problem is the contradicting need for architecture weights with high entropy at the beginning of the search (as not to favor non-trainable operations, such as pooling and Identity) and low entropy at the end (to approximate the collapse of the continuous distribution to a single operation). These limitations have led to several methods that try to alleviate them. For example, in [31], a more memory-efficient and stable version of DARTS is proposed. Other approaches try to prune the supernet more gradually [32], consider all incoming edges for a node simultaneously (as opposed to a per-node basis) [33], or make the hypergradient calculation (gradient of the architecture weights) more stable [34].

2.3 Evaluation Methods

Methods for evaluating intermediate solutions (networks generated during the search) and comparing NAS methods are essential to keep advancing the field. Candidate evaluation can significantly affect the computational cost of a search, while accurate method comparison can help decide on optimal methods for given applications. In this section, we present proposed solutions for both of these problems. These solutions have been utilized or further investigated in the scope of this thesis.

2.3.1 Candidate Evaluation

Candidate evaluation methods aim to evaluate candidate architectures during a search. Most search methods consider the relative performance of intermediate solutions to decide how to advance. For example, evolutionary methods utilize it as a fitness function, reinforcement learning approaches derive the discounted rewards from it, while differentiable methods back-propagate errors based on loss, strengthening the contribution of the best layer in the MixOp. The most straightforward approach for all of them is to utilize the original dataset to train and evaluate. Nonetheless, this is the most computationally expensive approach. Training a single deep learning architecture requires considerable time, and as such, many methods employ a number of parallel GPU workers to speed up the process, resulting in many GPU-days of computation [1, 8, 19]. Furthermore, relative performance is more important than absolute performance during the search phase, as the final architecture is usually trained from scratch after the search phase [10, 8, 19]. This realization has led to the utilization of proxy evaluation methods.

The most intuitive way to reduce the computational cost of evaluation during the search is to employ a smaller number of epochs (partial training) and fully train the final architecture afterward. This approach has been employed in various works with success, resulting in architectures that out-performed state-of-the-art human-designed networks [35, 36, 19, 2]. Although satisfactory empirical results have been observed, some researchers argue that when the proxy evaluation epochs differ significantly from the final evaluation epochs, it should not be employed [37]. To this extent, when network morphisms are used to evolve architectures, previously trained weights can be used to warm-start the training phase, reducing the epochs required to train networks to convergence [35].

The second most intuitive way to reduce candidate evaluation costs is to utilize fewer training instances. Utilizing an approach similar to transfer learning, researchers may opt to search for a neural architecture on a smaller dataset with similar characteristics to the target dataset and then train the final architecture on the target dataset [8, 32, 7]. In the spirit of reducing dimensions in the dataset, some researchers also reduce the dimensions of the candidate networks. This approach is prevalent in cell search spaces, as the number of cells and number of filters in layers can be scaled down for search and up for training, [19, 7, 38].

Finally, some approaches strive to avoid training interim solutions. This can be achieved in two ways; either with a model trained to predict the performance of a given architecture or by computationally evaluating the data-modeling abilities of a neural architecture. Predictive models can reduce the evaluation cost by several orders of magnitude, although a small subset of the architectures has to be trained and evaluated in order to train the predictive model [39, 40]. On the other hand, methods that evaluate the relative quality of architectures using untrained networks have been proposed [41]. This is achieved by measuring the overlap of activations between data points in a mini-batch of the original data.

2.3.2 Method Comparison

As NAS generating neural architectures is a complex process, and the results rely heavily on the pipeline chosen to load and pre-process data, implement and train networks, and guide the search, it is difficult to compare published results directly. Furthermore, results are usually reported in terms of final architecture performance, which introduces a second pipeline used to train the final architecture. Different data augmentation, optimizer hyper-parameters and extra regularization techniques can be implemented during the final train, often different from those used during the search. These problems are understood and documented by the NAS community, and an effort to establish a fair comparison of best practices has been made in the recent years [42].

One of the most robust approaches to compare methods is to utilize NAS benchmarks, aiming to provide a standardized architecture dataset. A well-known and extensive cell search space dataset is NASBench-101 [43]. Being a tabular dataset, it documents 423,624 unique cell architectures, and their performance on the CIFAR-10 dataset [20] for 4, 12, 36, and 108 training epochs. Only normal cells are considered, stacked in 3 groups of 3, with a down-sample layer between each group. Cells can have up to 7 nodes and 9 edges in a node-computational representation, with three-layer types; 3x3 convolutions and 1x1 convolutions, both followed by batch normalization and a ReLU activation layer, and 3x3 max-pooling. One node represents the input, and one node the output. As such, each cell has at most 5 computationally active nodes. Figure 2.11 depicts the outer skeleton, as well as a cell architecture extracted from the original paper. For each architecture-epoch pair, statistics concerning three separate (each one from scratch) training sessions are available; training, validation, and testing accuracy, the number of parameters in the final network and training time. Following this original work, there have been many other benchmarks released for NAS [44, 45, 46, 47].

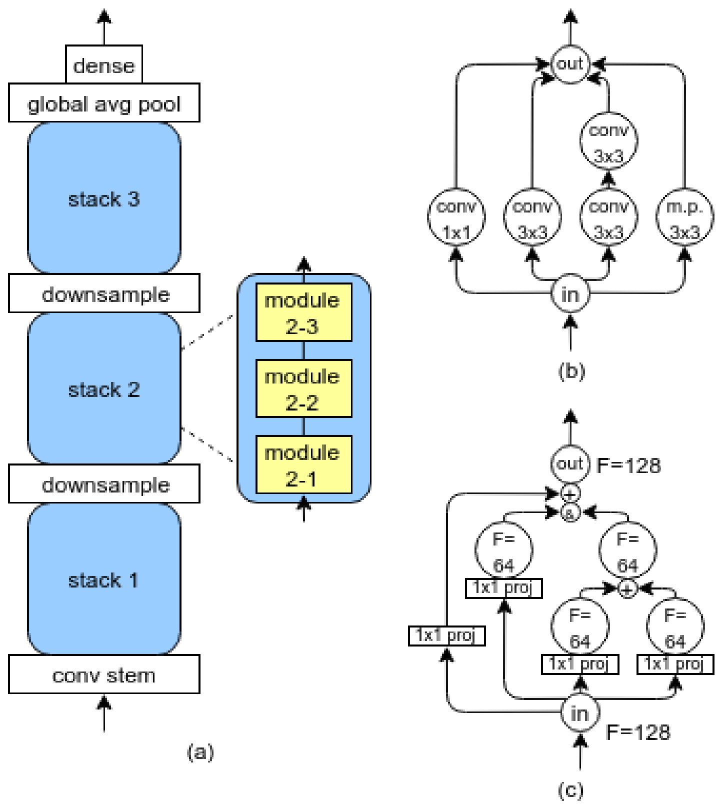


Figure 2.11: Neural architecture structure from [43]; (starting from left, clockwise) a. the outer skeleton, b. layer selection for a 7-node cell, c. lower-level calculation in the same cell.

Chapter 2 References

[1] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 11 2016. [Online]. Available: http://arxiv.org/abs/1611.01578

[2] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing.* Elsevier, 2019, pp. 293–312.

[3] F. Assunçao, N. Lourenço, P. Machado, and B. Ribeiro, "Denser: deep evolutionary network structured representation," *Genetic Programming and Evolvable Machines*, vol. 20, no. 1, pp. 5–35, 2019.

[4] E. Byla and W. Pang, "Deepswarm: Optimising convolutional neural networks using swarm intelligence," *UK Workshop on Computational Intelligence*, pp. 119–130, 2019. [Online]. Available: https://github.com/Pattio/DeepSwarm

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[7] M. S. Tanveer, M. U. K. Khan, and C.-M. Kyung, "Fine-tuning darts for image classification," in *2020 25th International Conference on Pattern Recognition (ICPR).* IEEE, 2021, pp. 4789–4796.

[8] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *Proceedings of the IEEE*

Computer Society Conference on Computer Vision and Pattern Recognition, pp. 8697–8710, 7 2018. [Online]. Available: http://arxiv.org/abs/1707.07012

[9] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," pp. 2423–2432, 2018.

[10] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *7th International Conference on Learning Representations, ICLR 2019*, 6 2018. [Online]. Available: http://arxiv.org/abs/1806.09055

[11] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[12] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 11 2018. [Online]. Available: http://arxiv.org/abs/1711.00436

[13] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[14] J. H. Holland, "Outline for a logical theory of adaptive systems," *Journal of the ACM (JACM)*, vol. 9, no. 3, pp. 297–314, 1962.

[15] P. A. Vikhar, "Evolutionary algorithms: A critical review and its future prospects," in *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*. IEEE, 2016, pp. 261–265.

[16] D. B. Fogel, L. J. Fogel, and V. Porto, "Evolving neural networks," *Biological cybernetics*, vol. 63, no. 6, pp. 487–493, 1990.

[17] H. Kitano, "Designing neural networks using genetic algorithms with graph generation system," *Complex systems*, vol. 4, pp. 461–476, 1990.

[18] D. E. Moriarty and R. Miikkulainen, "Forming neural networks through efficient and adaptive coevolution," *Evolutionary computation*, vol. 5, no. 4, pp. 373–399, 1997.

[19] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, 7 2019.

[20] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[21] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning.* PMLR, 2016, pp. 1928–1937.

[24] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[25] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *arXiv preprint arXiv:2006.02903*, 2020.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition.* Ieee, 2009, pp. 248–255.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning.* PMLR, 2015, pp. 448–456.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[29] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[30] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

[31] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong, "Pc-darts: Partial channel connections for memory-efficient architecture search," *arXiv preprint arXiv:1907.05737*, 2019.

[32] Y. Ding, Y. Wu, C. Huang, S. Tang, F. Wu, Y. Yang, W. Zhu, and Y. Zhuang, "Nap: Neural architecture search with pruning," *Neurocomputing*, 2022.

[33] Y. Jiang, C. Hu, T. Xiao, C. Zhang, and J. Zhu, "Improved differentiable architecture search for language modeling and named entity recognition," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3585–3590.

[34] M. Zhang, S. W. Su, S. Pan, X. Chang, E. M. Abbasnejad, and R. Haffari, "idarts: Differentiable architecture search with stochastic implicit gradients," in *International Conference on Machine Learning*. PMLR, 2021, pp. 12 557–12 566.

[35] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 2787–2794, 7 2018. [Online]. Available: http://arxiv.org/abs/1707.04873

[36] J. Liang, E. Meyerson, and R. Miikkulainen, "Evolutionary architecture search for deep multitask networks," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 466–473.

[37] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search," *arXiv preprint arXiv:1807.06906*, 2018.

[38] Q. Yao, J. Xu, W.-W. Tu, and Z. Zhu, "Efficient neural architecture search via proximal iterations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 6664–6671.

[39] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural architecture search with bayesian optimisation and optimal transport," pp. 2016–2025, 2018.

[40] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," *Advances in Neural Information Processing Systems*, vol. 2018-December, pp. 7816–7827, 8 2018. [Online]. Available: http://arxiv.org/abs/1808.07233

[41] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in *International Conference on Machine Learning.* PMLR, 2021, pp. 7588–7598.

[42] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *arXiv preprint arXiv:1909.02453*, 2019.

[43] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *International Conference on Machine Learning.* PMLR, 2019, pp. 7105–7114.

[44] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," *arXiv preprint arXiv:2001.00326*, 2020.

[45] A. Zela, J. Siems, and F. Hutter, "Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search," *arXiv preprint arXiv:2001.10422*, 2020.

[46] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, and E. Burnaev, "Nas-bench-nlp: neural architecture search benchmark for natural language processing," *arXiv preprint arXiv:2006.07116*, 2020.

[47] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," *arXiv preprint arXiv:2008.09777*, 2020.

Chapter 3

A Distributed Neural Architecture Search Framework

This chapter introduces Neural Operations Research and Development (NORD), a NAS framework developed and extensively utilized under the scope of this thesis. The implemented library is available at https://github.com/GeorgeKyriakides/nord

3.1 The complexity, Cost, and Speedup of NAS Pipelines

As already discussed, many critical points in a NAS procedure affect the result. This instability can be attributed to the extensive pipeline used to implement it, which increases the number of these points; dataset sources, preprocessing and augmentation, search space, search algorithm implementation, libraries used to implement the networks, evaluation hyper-parameters, and final architecture (which can differ from the NAS pipeline) training pipeline. This already complex NAS pipeline can be even more complicated when parallel or distributed methods are involved. Although NAS is generally a computationally intense procedure, the majority of time is spent searching and evaluating architectures. The rest of the NAS pipeline components require significantly fewer resources. For example, a conventional DARTS run may require close to 4 days on a GTX1080Ti GPU to find the optimal architecture for CIFAR10 [1], while downloading CIFAR10 over a 100Mbps connection requires roughly 20 seconds and loading the dataset

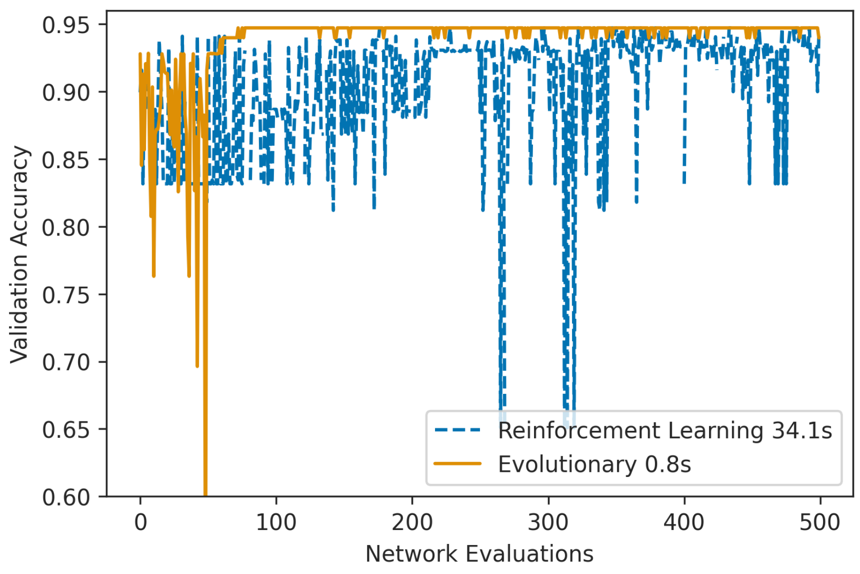


Figure 3.1: Comparison of run times (search algorithm only) and network validation accuracy between a reinforcement learning and an evolutionary algorithm.

from an average SSD (500 MB/s read speed over SATA III) requires less than 2 seconds. Training the final architecture is usually not considered part of the computational cost of NAS (as human-designed architectures also undergo this procedure).

Figure 3.1 depicts the runs on NASBench-101 of a regularized evolution algorithm and a reinforcement learning algorithm, similar to [2] and [3] respectively. The time required for the search logic itself (i.e., managing the evolved population or the RL agent) is 0.8 seconds for the evolutionary algorithm and 34.1 seconds for the reinforcement learning agent. Compared to the time required to load the data (assuming the search was conducted on CIFAR-10, which NASBench-101 utilizes), which is almost 2 seconds, the search times for reinforcement learning are considerably higher. Compared to the time required to train the networks, which is close to 30,000 seconds, or 23 hours and almost 20 minutes, the cost of search algorithms is negligible.

As NAS has high computational and wall-clock time costs, speeding up the process is desirable. The most straightforward approach to accelerating any NAS method with parallel and distributed computing is to focus on network evaluation. Either data parallelization (distributing each batch to N GPU workers) as distributed SGD or model parallelization (distributing the network to N workers) can be employed. As model parallelization is usually more difficult to implement successfully, more architecture-specific and more communication intense [4], data parallelization is preferred. The second most straightforward approach is to parallelize the search algorithm. Note that search algorithm parallelization aims at evaluating *multiple networks* concurrently, rather than a single network. Figure 3.2 shows the NAS pipeline and the parallelizable components in green. Blue components can be parallelized, but their relative computational cost is negligible. Red components cannot be parallelized (Search Space, Network Compilation) or do not contribute to the overall NAS cost (Final Architecture Training). For gradient-based approaches, the search algorithm and evaluation overlap. In these cases, parallelizing the search algorithm, in fact, aims at evaluating the current architecture in a data-parallel fashion.

3.2 Need for a Framework

Although it can provide significant speedup, Parallelizing NAS methods introduces two major problems. It increases the complexity of implementing a NAS solution, as well as the complexity of ensuring reproducibility, an already prevalent problem in NAS literature [5]. Both of these problems can be alleviated by establishing a NAS Framework. Benchmark datasets ([6], [7], [8]) provide a means to fairly compare algorithms, but cannot impose restrictions on how these algorithms may be applied to other datasets. In this thesis, we propose a NAS framework and library called NORD (Neural Operations Research and Development), based on directed graphs, which represent network or cell architectures and PyTorch [9]. The aims of NORD are the following:

- Provide an easy to understand programming model.
- Efficiently define, optimize, and evaluate an architecture.
- Standardize the NAS pipeline.

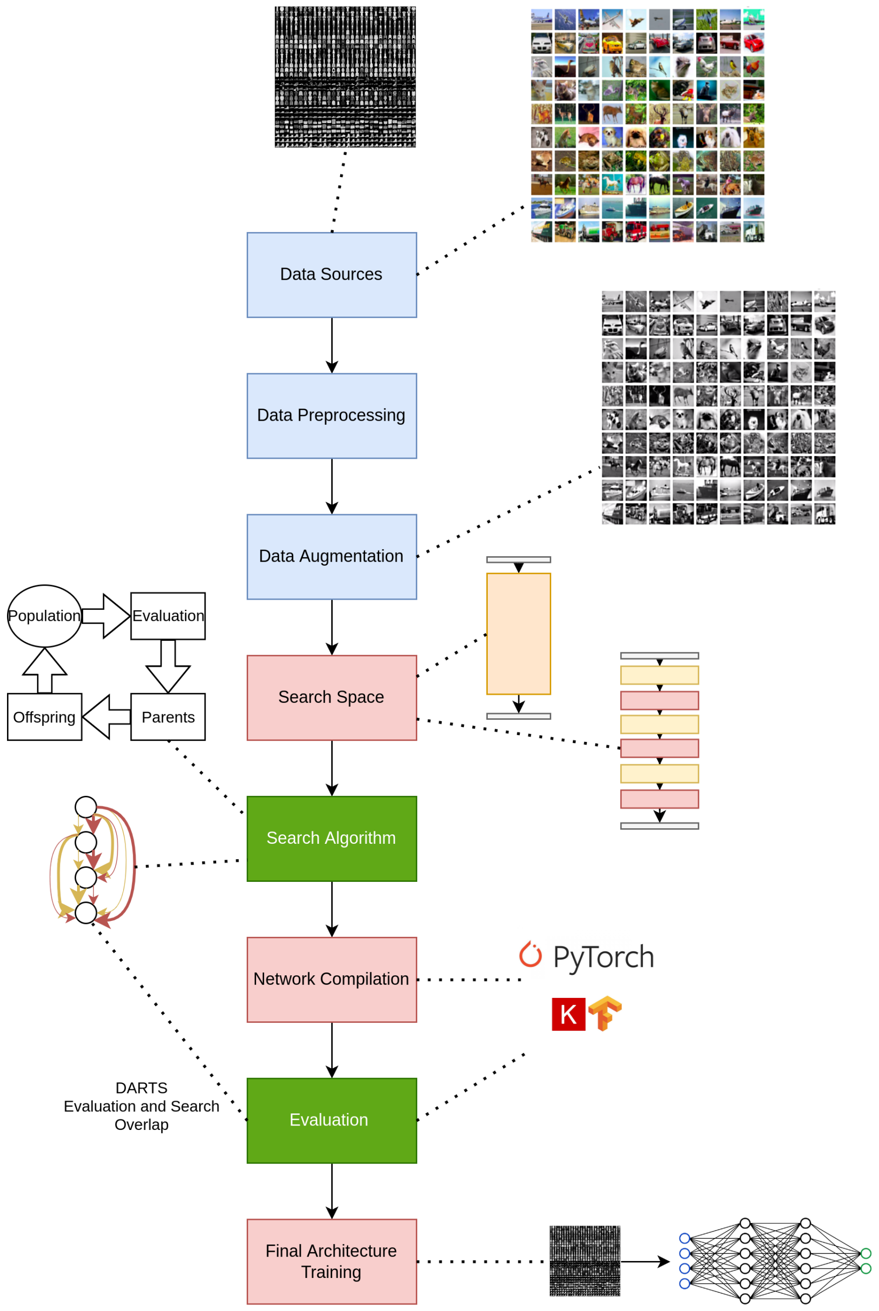


Figure 3.2: NAS pipeline, parallelizable components in green. Blue components have negligible relative computational cost, while red components do not contribute to overall cost or cannot be parallelized.

- Hide network compilation from the user.
- Allow for relatively easy distributed evaluation.
- Allow for a fair comparison between NAS methods for any dataset.

3.2.1 NORD Components

NORD employs a number of basic components based on a standard NAS pipeline; architecture descriptors, architecture evaluators, data curators and distributed evaluation environments. In this subsection, we present these components and their primary usage.

Descriptors

Descriptors allow a user-friendly interface to define and alter architectures. Descriptors retain information in a node computational (unlike DARTS' edge computational) format. As the name implies, a descriptor does not contain instantiated layers and connections. Instead, only a description of the architecture is retained. Each node's layer class and parameters are saved along with an optional layer name. Layers can be added sequentially, where each new layer's input is the output of the previous layer. Alternatively, a layer can be added to the descriptor without any connection. The layer can later get connected to any other layer (either an input or an output) by specifying the names of the source and destination layers. Furthermore, connections can be removed, again using layer names. When using Nord, the NeuralDescriptor class can be imported from nord.neural_nets. An example of describing the network in Figure 3.3 is provided with PyTorch layers:

```python
from nord.neural_nets import NeuralDescriptor
import torch.nn as nn

# Define layer presets
conv = nn.Conv2d
conv_params = {'in_channels': 3,
```

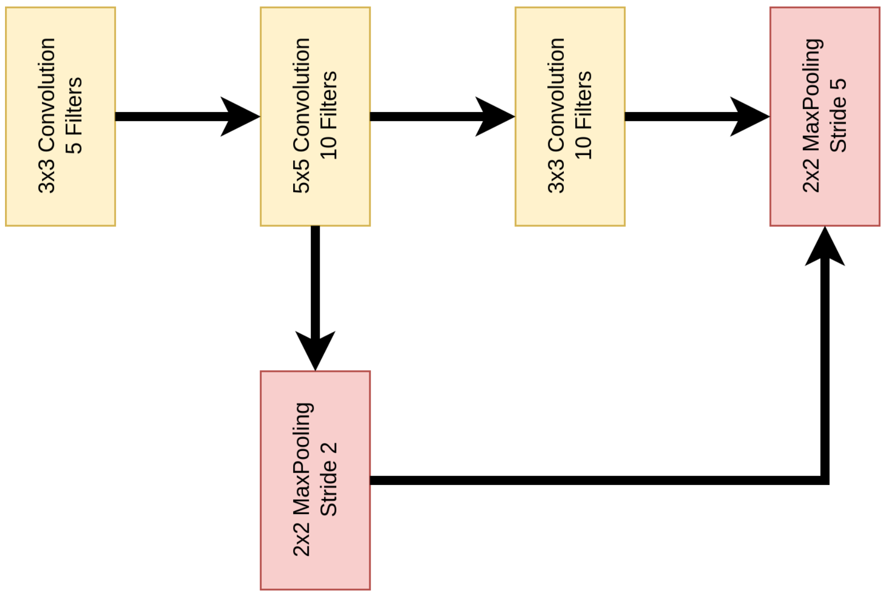


Figure 3.3: The network implemented in 3.1.

```
               'out_channels': 5, 'kernel_size': 3}

conv_2_params = {'in_channels': 5,
                 'out_channels': 10, 'kernel_size': 5}

conv_3_params = {'in_channels': 5,
                 'out_channels': 10, 'kernel_size': 3}

pool = nn.MaxPool2d
pool_params = {'kernel_size': 2, 'stride': 2}

pool2_params = {'kernel_size': 2, 'stride': 5}

d = NeuralDescriptor()

# Add layers and give them names
d.add_layer(conv, conv_params, 'conv')
d.add_layer(conv, conv_2_params, 'conv2')
d.add_layer(conv, conv_3_params, 'conv3')
d.add_layer(pool, pool_params, 'pool1')
d.add_layer(pool, pool2_params, 'pool2')

```

```
# Add Connections using names
d.connect_layers('conv', 'conv2')
d.connect_layers('conv2', 'conv3')
d.connect_layers('conv3', 'pool2')
d.connect_layers('conv2', 'pool1')
d.connect_layers('pool1', 'pool2')
```

Listing 3.1: Descriptor example

Evaluators

Evaluators allow the network compilation of an architecture described by a NeuralDescriptor and its evaluation on a specified dataset. They are responsible for ensuring that a viable architecture is compiled consistently and evaluated in a reproducible way. In addition, evaluators ensure that many technical details are hidden from the end-user, which can significantly ease the development and implementation of NAS methods. An example of such details is handling multiple inputs to a layer. As architectures are generated, a single layer may receive inputs from layers where the dimensions are not aligned.

For example, a 2d convolution layer A may have an output size of 15X15X10 (Height X Width X Channels), while another layer B may have an output size of 3X3X50. If these two layers' outputs are declared as the input of a third layer C, with minimum viable input (e.x. due to kernel size) of 10X10X18, the inputs must be scaled and combined. NORD scales the Height and Width dimensions in such scenarios by first determining the minimum viable size (MVS). If a layer's output is different from the MVS, its Height and Width dimensions are up-scaled with interpolation to the MVS. Following, the channels are scaled utilizing a 1X1 convolution layer, with a number of filters matching MVS' channels and the inputs are summed. As such, C's inputs would have dimensions equal to MVS.

An alternative approach would be to find the *Maximum Viable Size* and upscale all layers and channels. Furthermore, to ensure that information is not lost, a concatenation could be applied instead of summing the inputs. Although this approach was implemented in the first iteration of NORD development, internal network dimensions of compiled networks were growing rapidly, leading to shallow networks with many parameters. It is easy to see why; assuming a layer D with output size 100X100X60, any other layer which gets input from D would have an input size of at least 100X100X60. If D is one of the network's first layers (close to the input layer), it is highly likely to feed many layers, imposing a high input size. Furthermore, as most NAS methods utilize convolutions with relatively small kernels (3X3 and 5X5 being the most common), the outputs of these layers will also be high.

A third approach for handling internal sizes, implemented in NORD, concerns fixed filter numbers throughout the networks. This is an approach used in various papers but was first implemented in the NASNet cell search space paper [3]. Various experiments later presented in this thesis have successfully used this approach for image and graph data. NORD currently implements four different evaluator classes able to evaluate descriptors through the (descriptor_evaluate) function:

LocalEvaluator Local evaluators are the most simple implementation of an evaluator. Data is loaded locally, and a descriptor is evaluated by compiling the corresponding network and training it for a given number of epochs. An optimizer class and its parameters must be passed to the constructor when instantiating an evaluator. When evaluating a descriptor, the final loss function value, a dictionary with metrics and the total time required to evaluate the network are returned. A number of datasets are implemented, such as CIFAR-10 [10], and Fashion-MNIST [11], as well as graph datasets presented later in the thesis. Custom datasets are also supported. An example of using LocalEvaluators is provided below:

```
from nord.neural_nets import LocalEvaluator, NeuralDescriptor
evaluator = LocalEvaluator(optimizer_class=opt.Adam,
    optimizer_params={})
dataset = 'cifar10'

# in_channels are 3 for the first layer, as CIFAR-10
# is in RGB format
conv_params = {'in_channels': 3,
                'out_channels': 5, 'kernel_size': 3}
conv_2_params = {'in_channels': 5,
                 'out_channels': 10, 'kernel_size': 3}
pool = nn.MaxPool1d
pool_params = {'kernel_size': 2, 'stride': 2}

d = NeuralDescriptor()
d.add_layer(conv, conv_params)
d.add_layer_sequential(conv, conv_2_params)
d.add_layer_sequential(pool, pool_params)

loss, metrics, total_time = evaluator.descriptor_evaluate(
    descriptor=d, epochs=2, dataset=dataset)

# Print the results

print('Train time: %.2f' % total_time)
print('Test metrics: ', [(key+': %.2f' % value)
                         for key, value in metrics.items()])
print('Test loss: %.2f' % loss)
```

Listing 3.2: Local evaluator example

DistributedEvaluator Distributed evaluators are responsible for partitioning training data amongst workers and coordinating the training process. Their API is similar to local evaluators, allowing an almost seamless transition. However, data communication is achieved through PyTorch's distributed API, and as such, efficiently utilizing them requires setting up the distributed environment correctly.

NASBench_101Evaluator and NATSBench_Evaluator Benchmark evaluators are also implemented in NORD. When testing a NAS implementation on a benchmark dataset, descriptors must contain the benchmark dataset's predefined layer strings as layer types. NASBench_101Evaluator implements the NASBench-101 dataset [7], while NATSBench_Evaluator implements the NASBench-201 dataset [6]. Both evaluators provide a list of available operations through the **get_available_ops** function. Note that input and output node names are not in the list. Instead, NASBench-101 defines the names 'input' and 'output', while NASBench-201 defines 'IN' and 'OUT', respectively. Examples for both evaluators are provided below.

```
## NASBench-101 example

from nord.neural_nets import NASBench_101Evaluator,
    NeuralDescriptor

# Instantiate the evaluator
evaluator = NASBench_101Evaluator()
# Instantiate a descriptor
d = NeuralDescriptor()

# See the available layers (ops)
# for NASBench-101
layers = evaluator.get_available_ops()
print(layers)

# Add NASBench-101 Layers connected
# sequentially
d.add_layer('input', None, 'in')
d.add_layer_sequential(layers[0], None, 'layer_1')
d.add_layer_sequential(layers[2], None, 'layer_2')
d.add_layer_sequential('output', None, 'out')


# Add Connections
d.connect_layers('layer_1', 'out')

# Get the validation accuracy and training time
val_acc, train_time = evaluator.descriptor_evaluate(
    d, acc='validation_accuracy')

## NASBench-201 example

from nord.neural_nets import NATSBench_Evaluator,
    NeuralDescriptor

```

```
# Instantiate the evaluator
ne = NATSBench_Evaluator()

# See the available layers (ops)
# for NASBench-201
layers = evaluator.get_available_ops()
print(layers)

# Add NASBench-201 Layers connected
# sequentially
descriptor = NeuralDescriptor()
descriptor.add_layer("IN", None, "IN")
descriptor.add_layer_sequential("nor_conv_3x3", None, "1")
descriptor.add_layer_sequential("nor_conv_1x1", None, "2")
descriptor.add_layer_sequential("avg_pool_3x3", None, "3")

# Note that does NOT work with 5 nodes!
# If we remove the following comment the evaluator
# will reject the descriptor
# descriptor.add_layer_sequential("avg_pool_3x3", None, "4")

descriptor.add_layer_sequential("OUT", None, "OUT")


descriptor.add_layer("skip_connect", None, "4")
descriptor.connect_layers("1", "4")
descriptor.connect_layers("4", "OUT")


descriptor.add_layer("skip_connect", None, "5")
descriptor.connect_layers("IN", "5")
descriptor.connect_layers("5", "3")


descriptor.add_layer("nor_conv_3x3", None, "6")
descriptor.connect_layers("IN", "6")
descriptor.connect_layers("6", "OUT")

# Get the validation accuracy and time cost
```

```
ne.descriptor_evaluate(descriptor, metrics=['validation_accuracy'
    , 'time_cost'])
```

Listing 3.3: Benchmark evaluators examples

Environment

The Environment class (from nord.distributed.environment) is responsible for setting up the distributed environment and incorporating a DistributedEvaluator. The class is implemented as a context manager, enabling the quick translation of locally-executed NORD code into distributed. Using PyTorch's distributed API and NVIDIA's NCCL backend ensures quick collective communications between nodes. It uses a Master-Worker architecture, where each worker instantiates a DistributedEvaluator and waits for the Master process to send an architecture encapsulated in a descriptor and a dataset to start evaluating. Environment follows the Evaluators API and provides a descriptor_evaluator function. The Environment's __enter__ process ensures the correct distributed environment setup, while its __exit__ process ensures the correct process group shutdown. Finally, a DistributedConfig class is implemented, containing information regarding the process group; the master node's IP and communication port, world size and world rank are included.

The code enclosed under the Environment context manager is executed solely by the master process. This restriction ensures no worker overhead for executing the NAS logic, and only evaluation is distributed. This also avoids situations where a worker lags due to search method logic or where it tries to execute code depending on data available only to the master process (for example, a reinforcement learning agent's prediction network). A high-level overview of the Environment's logic and communications is depicted in Figure 3.4, while a usage example is provided directly below.

```

from nord.neural_nets.distributed.environment import Environment

# Configuration, Master listens at localhost:1234 and there are 4
    GPU workers.
```

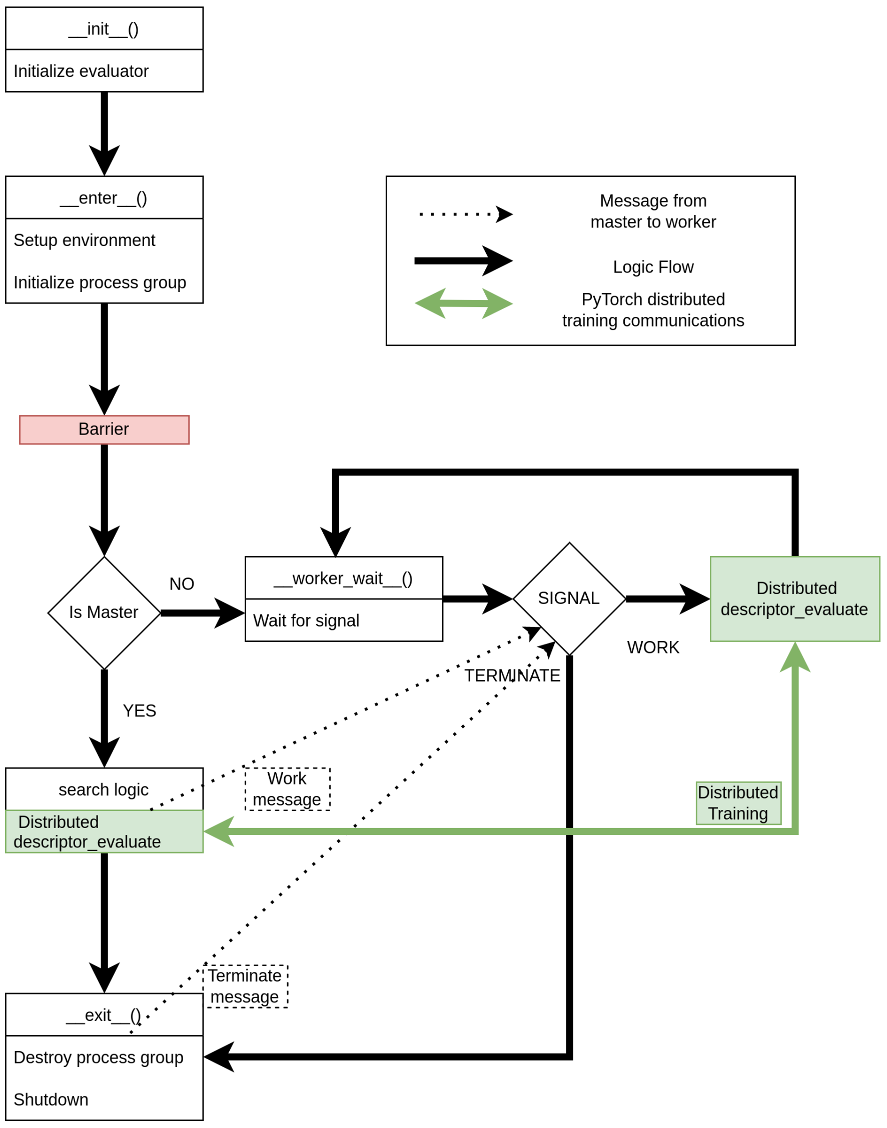


Figure 3.4: Logic and communication under the Environment context manager.

```
config = DistributedConfig('127.0.0.1', '1234', world_size=4,
    world_rank=0)

# Assuming a valid descriptor is provided
with Environment(config) as e:
    loss, metrics, total_time = e.descriptor_evaluate(
        descriptor=descriptor, epochs=2, dataset='cifar10)
```

Listing 3.4: Environment example

Environments can be utilized in two ways with MPI and other distributed interfaces. First, instantiating a DistributedConfig object can be done with MPI by copying each worker's world size and rank attribute to the DistributedConfig object. The Master's IP address and port can also be broadcast during runtime or implemented as program arguments. The second approach entails parallelizing the search method. A good example is the concurrent evaluation of a genetic algorithm population. Assuming N nodes, each with G GPUs and given sufficient problem sizes, instead of having NXG independent network evaluations active at any time, the distributed environment allows the utilization of N data-parallel network evaluations. Each node is a local master and orchestrates G local GPU workers.

Data Curators and Configurations

As the name implies, data curators are responsible for curating data, while the Configurations class is responsible for configuring a NAS pipeline tied to a specific dataset (implemented as a singleton). All datasets can be retrieved with the get_{dataset name} function, where a percentage parameter determines what percentage of the dataset will be used for training, thus implementing partial training. NORD curators provide 7 distinct datasets (excluding the benchmarking datasets), depicted in Table 3.1.

Configurations contain information for a number of parameters essential to the pipeline, provided in the list below:

- Number of outputs (equal to 1 for regression, number of classes for classification).
- Input shape for each dataset's channel (Width and Height).
- Number of channels.
- Optimization criterion for the dataset (i.e. loss function).
- Metrics that must be calculated, such as accuracy or mean squared error.

Table 3.1: NORD Datasets

Dataset	CIFAR-10	Fashion-MNIST	NASBench-101 Architectures	Fashion-MNIST Architectures	Activity Recognition
Publication	[10]	[11]	[12]	[12]	[13]
Name String	cifar10	fashion-mnist	{graph-nasbench3, graph-nasbench1}	{graph-f-mnist3, graph-f-mnist1}	activity_recognition
Type	matrix (Image)	matrix (Image)	graph (Neural Architectures)	graph (Neural Architectures)	matrix (Time Series)
Outputs (Classes)	10	10	{3,1}	{3,1}	7
Input Shape	32X32	28X28	5 (Size of feature vector)	5 (Size of feature vector)	52
Channels	3	3	1	1	1
Criterion	Cross Entropy Loss	Cross Entropy Loss	{KL Divergence, Margin Ranking Loss}	{KL Divergence, Margin Ranking Loss}	Cross Entropy Loss
Metrics	Accuracy	Accuracy	Kendall's τ, Spearman's ρ	Kendall's τ, Spearman's ρ, Accuracy	Accuracy

- Whether dimension keeping (retaining the same number of channels for all layers) is applied.
- Data loading functions (from data curators)
- Dense part (head) for the compiled networks.
- Input data organization (graph or matrix-like).

A relatively interesting parameter is the dense part of compiled networks. Architectures generated (either cell or global) usually concern the network's convolutional/feature extraction part. The head of the network (final layers leading to the output) is pre-determined, possibly as a part of the search space. Nonetheless, it can also affect the performance of the generated networks. As such, when an Evaluator compiles the network, it ensures that all networks utilize the same head, which is pre-defined for a network. Configurations provide an API to specify custom datasets through the add_regression_dataset, add_classification_dataset and add_dataset functions. An example of adding a custom dataset (using PyTorch's MNIST dataset) is provided below:

```python
import torchvision
from torchvision import transforms
from nord.configurations.all import Configs
from nord.neural_nets import LocalEvaluator

# Percentage dictates what percentage of the trainset
# will be used while training (not train/test split percentages.
# This is not implemented in
# this toy example.
def get_mnist(percentage: float = 1):
    print('Loading MNIST.')
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    trainset = torchvision.datasets.MNIST(root='./my_data/mnist',
                                          train=True,
                                              download=True,
                                              transform=transform)

    trainloader = torch.utils.data.DataLoader(trainset,
   batch_size=128,
                                                   num_workers=0,
                                                   shuffle=True)

    testset = torchvision.datasets.MNIST(root='./my_data/mnist',
                                             train=False,
                                             download=True,
                                             transform=transform)

    testloader = torch.utils.data.DataLoader(testset, batch_size
   =128,
                                                  shuffle=False,
   num_workers=0)

    classes = ('0', '1', '2', '3', '4',
               '5', '6', '7', '8', '9')

    return trainloader, testloader, classes



# Instantiate the Configs singleton and add the dataset
conf = Configs()
# data_organization is optional, defaults to 'matrix' i.e. image
conf.add_classification_dataset(name='MNIST', num_classes=10,
   input_shape=(
    28, 28), channels=1, data_load=get_mnist)

# Assuming a valid descriptor in the descriptor variable
evaluator = LocalEvaluator(torch.optim.Adam, {}, verbose=True)
evaluator.descriptor_evaluate(d, epochs=2, dataset='MNIST')
```

Listing 3.5: Custom dataset example

3.3 NORD NAS Examples

This section provides various example implementations for benchmark and non-benchmark datasets. The aim is to provide a basic understanding of utilizing NORD and its abilities.

3.3.1 A simple evolutionary algorithm

In this example, we implement a simple evolutionary algorithm similar to the one proposed in [2]. The python file with the corresponding implementation is reproducible_genetic_algorithm_example_nasbench.py. First, we instantiate the NASBench-101 evaluator and get the available operations. Following, we define two helper functions; the first adds a random connection to a descriptor, while the second mutates a descriptor, adding either a random connection between nodes or replacing a random connection with a node.

```python
evaluator = NASBench_101Evaluator()
ops = evaluator.get_available_ops()

def add_random_connection(descriptor: NeuralDescriptor):
    # Get a source, but don't get output as source node [:-1]
    from_ = np.random.choice(list(descriptor.connections.keys())[:-1])
    # NORD names layers with a number prefix. Get the source node number
    # and find possible output nodes that have a greater number as prefix
    from_number = from_.split('_')[0]
    avail_to_layers = [ x for x in descriptor.layers.keys() if int(from_number) < int(x.split('_')[0])]
    # If there are available output layers, choose one at random and add the connection
    if len(avail_to_layers) > 1:
        to_ = np.random.choice(avail_to_layers)
        descriptor.connect_layers(from_, to_)

    return descriptor
```

```python


def mutate(descriptor: NeuralDescriptor):
    r = np.random.uniform()
    try:
        # If we have the maximum number of hidden layers (+2 to account for input-output layers) we can only add connections
        if r < add_connection_prob or len(descriptor.layers) == max_layers+2:
            descriptor = add_random_connection(descriptor)
        # If we have less layers and the random number generator dictates to add
        # a layer, proceed
        elif r < add_node_prob+add_connection_prob:
            # Get source node, but again don't get the output layer as source node
            from_ = np.random.choice(list(descriptor.connections.keys())[:-1])
            # Get all of its connections and choose one at random to replace
            to_ = np.random.choice(list(descriptor.connections[from_]))
            # Get a random layer
            layer_index = np.random.choice(available_layers)
            # Name it with a prefix that is larger than the source node
            # and smaller than the destination node. Getting the average
            # of the source and destination prefixes is sufficient.
            layer_name = (int(from_.split('_')) + (to_.split('_')))/2
            layer_name = str(layer_name)+'_Layer'
            layer_name = descriptor.add_layer(ops[layer_index], {}, layer_name)

            # Disconnect the original source and destination nodes
            descriptor.disconnect_layers(from_, to_)
            # Connect them to the new layer
            descriptor.connect_layers(from_, layer_name)
            descriptor.connect_layers(layer_name, to_)
        # Ensure that the descriptor's last layer name points
        # to the actual output layer. Adding a layer
        # changes the last_layer name, so we have to reset it
        descriptor.last_layer = OUTPUT_LAYER_NAME
    except Exception:
        pass

    return descriptor
```

Following, we define a simple evaluation function that handles the exception thrown when an invalid architecture is passed and returns 0 in that case, as well as a named tuple to hold individuals' descriptors and fitness values, while also defining the population initialization and evolution:

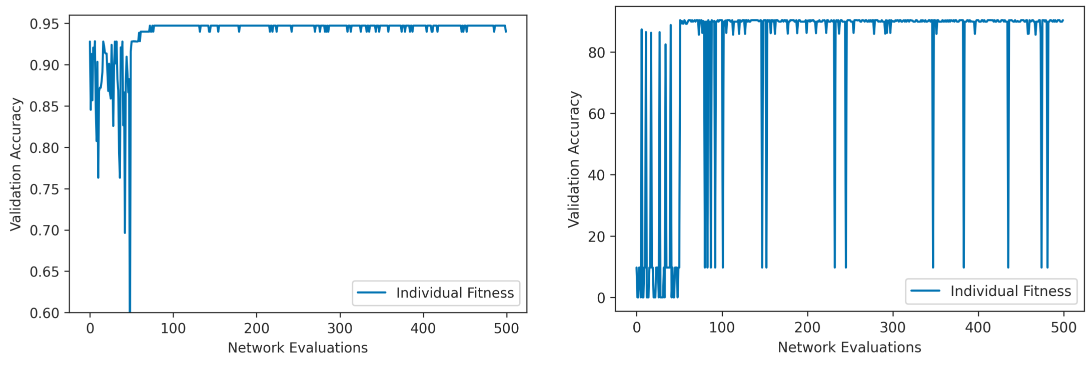
```
OUTPUT_LAYER_NAME = '9999'
IN_LAYER = 'input'
OUT_LAYER = 'output'

def evaluate(descriptor):
    fitness = 0
    total_time = 0
    try:
        fitness, total_time = evaluator.descriptor_evaluate(
            descriptor)
    except Exception as e:
        pass
    return fitness

Individual = namedtuple('Individual', 'descriptor fitness')

for i in range(initial_population_size):
    d = NeuralDescriptor()
    # How many layers this individual will initially have.
    # We require at least 1
    this_layers = 1+np.random.choice(max_layers-1)
    # Add the first (input) layer
    d.add_layer_sequential(IN_LAYER, {})
    # For the number of layers that this individual will have
    # add a random sequential layer
    for _ in range(this_layers+1):
        layer_index = np.random.choice(available_layers)
        layer_name = d.add_layer_sequential(ops[layer_index], {})
        # Give a 1/7 probability to add a second connection
        if np.random.uniform() < 1/7:
            d = add_random_connection(d)
    # Add the output layer and evaluate
    d.add_layer_sequential(OUT_LAYER, {}, OUTPUT_LAYER_NAME)
    f = evaluate(d)
    population.append(Individual(d, f))


while generation < evolutions:

    generation += 1
    # Get the candidate parents
    sample = np.random.choice(len(population), size=sample_sz)
    sample = [population[x] for x in sample]
    # Get their fitnesses
    fits = [x.fitness for x in sample]
    # Get the best parent
    parent = np.argmax(fits)
    parent = sample[parent]
    # Copy the parent's descriptor, mutate and evaluate
    offspring = deepcopy(parent)
    offspring = mutate(offspring.descriptor)
    fitness = evaluate(offspring)
    # Add the offspring to the population and discard
    # the oldest individual
    population.append(Individual(offspring, fitness))
    population.pop(0)
```

Listing 3.6: Regularized evolution for NASBench-101

The above algorithm can find one of the top NASBench-101 architectures in less than 50 evolution cycles. Changing to NASBench-201 is relatively easy 3.7, although the algorithm is tuned for the NASBench-101 search space and, as such, does not provide the same performance. As it can be seen from Figure 3.5, a number of invalid architectures are generated during population initialization. Nonetheless, the algorithm finds a relatively well-performing architecture, given its insufficient mutation rules.

```
evaluator = NATSBench_Evaluator()
IN_LAYER = 'IN'
OUT_LAYER = 'OUT'
```

Listing 3.7: Changes to apply regularized evolution to NASBench-201



(a) NASbench-101 (b) NASbench-201

Figure 3.5: Performance of regularized evolution on NASBench-101 and NASBench-201.

3.3.2 A simple reinforcement learning algorithm

In this example, we provide an implementation of a reinforcement learning agent for the NASBench-101 dataset. Contrary to [14], [3] we do not use a RNN controller. Instead, we use a fully connected network, taking advantage of the limited size of cells in NASBench-101, resulting in small binary representations which we use as states. Furthermore, we avoid the need to train recursive networks, resulting in faster training times.

The controller network receives an input with a size of 21 (possible connections between nodes) + 24 (6 layers, encoded in 4-bit one-hot vectors) bits, for a total of 45. We employ two output layers, one for layer type selection (4 output neurons, one for each layer plus one for the output layer) and one for input selection (7 neurons, one for each layer before the output, plus one for no-connection). The input selection layer utilizes an attention-like mechanism; a softmax is applied over the outputs, and the two neurons with the highest activation are selected. Their position indicates the inputs for the selected layer (with the seventh layer indicating no connection). The details of implementing a reinforcement learning agent's controller are outside the scope of this thesis and, as such, are omitted. Nonetheless, the full implementation can be found in the reinforcemnet_learning_attention_like_nasbench.py file.

To conduct NAS with a reinforcement learning method, we must first define our agent's environment:

```python

# +1 as we can stop before maximum number of layers
n_actions = ops_number+1

class NASEnv:

    def __init__(self):
        # Get the layers available to the search space
        self.avail_layers = evaluator.get_available_ops()
        self.layer_codes = {}

        # Create one-hot vector codes for each layer
        for i in range(len(self.avail_layers)):
            code = np.zeros(ops_number+1)
            code[i] = 1.0
            self.layer_codes[self.avail_layers[i]] = code

        # Create a code for the input layer as well
        self.layer_codes[INPUT_LAYER] = np.array([0, 0, 0, 1])
        # Create the initial descriptor with only an input layer
        self.descriptor = NeuralDescriptor()
        self.descriptor.add_layer_sequential(INPUT_LAYER, {})

    # Reset function called after an episode ends
    def reset(self):
        self.descriptor = NeuralDescriptor()
        self.descriptor.add_layer_sequential(INPUT_LAYER, {})

    # Return the state of the environment as a torch tensor
    def get_state(self):

        # Create a temporary descriptor with an output layer
        # in order to create the representation
        tmp_desc = deepcopy(self.descriptor)
        tmp_desc.add_layer_sequential(OUTPUT_LAYER, {}, 'tmp_out'
)
        # Get the descriptor as a matrix-operations
representation
        # matrix contains the connections between layers
        # while ops contains the type (as string) of each layer
        matrix, ops = evaluator.descriptor_to_matrix(tmp_desc)

        # Create one-hot representation of layer types
        layers = np.zeros(6*(ops_number+1))
        # Don't need the output layer, as it is always there
        for i in range(len(ops)-1):
            op = ops[i]
            code = np.array(self.layer_codes[op])
            layers[i*(ops_number+1):(i+1)*(ops_number+1)] = code

        # Transfer the connection matrix to a 7X7 connection
matrix
        # The original may be smaller than 7X7, due
        # to having less
        connections = np.zeros((7, 7))

        matrix = np.array(matrix)
        # Copy all but last column
        connections[0:matrix.shape[0], 0:matrix.shape[1]-1] =
matrix[:, :-1]

        # Copy last original column to last column of connections
,
        # as it refers to the output layer
        connections[0:matrix.shape[0], -1] = matrix[:, -1]

        # Get the upper triangle indices of the 6X6 matrix
        # We don't need the full 7X7 matrix as the first column
        # and last row are always zero
        inds_x, inds_y = np.triu_indices(6)
        # Move columns to the right as first column is zeros
        inds_y += 1
        connections = connections[(inds_x, inds_y)]

        # Return the concatenated one-hot layer representations and connections
        return torch.tensor(np.concatenate([layers.reshape(1, -1), connections.reshape(1, -1)], axis=1)).unsqueeze(1)

    # For a given action perform a step, update the state
    # and return a flag indicating if it is a terminal state (only when
    # the agent asks to stop or the maximum number of nodes has been
    # added) and the reward (zero if the state is not terminal,
    #  as we only evaluate networks that are in terminal states)
    def step(self, action):


        current_layers_no = len(self.descriptor.layers)

        # Action contains a tuple of layer selection
        # and layer connections
        layer, layers_in = action
        # Layer connections indicate always two layers
        # No-connection is implemented as selecting the output
        # layer for input
        in1, in2 = self.get_connections(layers_in)
        reward = 0.0
        done = False
        # If a terminal state
        if current_layers_no == 6:
```

```python
            # Add an output layer and ask the evaluator
            # to evaluate the descriptor
            self.descriptor.add_layer_sequential(OUTPUT_LAYER, {})
            reward, time = evaluator.descriptor_evaluate(self.descriptor)
            done = True
        # No terminal state, add the selected layer
        else:

            # Initializing with output layer
            layer_params = (OUTPUT_LAYER, {})
            # If the agent has not asked to stop
            if layer < len(self.avail_layers):
                layer_params = (self.avail_layers[layer], {})

            # Add the layer and get the generated name
            layer_name = self.descriptor.add_layer(*layer_params)

            # If in1 does not indicate no-connection,
            # add the requested connection
            if in1 != NO_CONNECTION:
                in1_layer = list(self.descriptor.layers.keys())[in1]
                self.descriptor.connect_layers(in1_layer, layer_name)

            # Same for in2 selection
            if in2 != NO_CONNECTION:
                in2_layer = list(self.descriptor.layers.keys())[in2]
                self.descriptor.connect_layers(in2_layer, layer_name)
            # If the agent asked to stop, evaluate the network and
            # set the terminal state flag
            if layer == len(self.avail_layers):
                reward, time = evaluator.descriptor_evaluate(self.descriptor)
                done = True

        return reward, done

    # Helper function to get selected connections from the
    # model's outputs
    def get_connections(self, layers_in):
        # Get current layers number
        current_layers_no = len(self.descriptor.layers)
        # Transfer selection from GPU memory to CPU and from
        # pytorch tensor to nunmpy array
        layers_in = layers_in.cpu().numpy().reshape(-1)
        # Sort by value and get the two highest value indices
        sorted_inds = np.argsort(layers_in[:current_layers_no])

        in1 = sorted_inds[-1]
        in2 = NO_CONNECTION
        # If there are enough layers, get
        # the second connection
        if len(sorted_inds) > 1:
            in2 = sorted_inds[-2]
        # Return the indices
        return in1, in2
```

Listing 3.8: Environment for NASBench-101

After the environment is implemented, finding an architecture consists of iterating over a number of episodes, taking actions, evaluating the resulting states, and training the controller. The results of 500 episodes are depicted in Figure 3.6. Compared to the evolutionary algorithm, reinforcement learning is more unstable, while it fails to generate the same quality of architectures. Figure 3.1 was generated by utilizing the implementations presented in this section, which better compares the differences between these two approaches.

```

# Using Double Deep Q-Networks algorithm
# we have two networks for the agent:
# policy_net which dictates the agent's moves
# and target_net which predicts the value of each action
```

```

for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    # Get the initial state
    state = env.get_state()
    for t in count():
        # Select and perform an action (layer type and connections)
        action = select_action(state)
        # Get the reward
        reward, done = env.step(action)

        # Transfer the reward to cuda
        reward = torch.tensor([reward], device=device)

        # Observe new state
        if not done:
            next_state = env.get_state()
        else:
            next_state = None

        # Get the selected connections
        in1, in2 = env.get_connections(action[1])
        in1 = in1
        in2 = in2
        # Store the transition in memory
        memory.push(state, action[0], in1, in2, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()

        # If a terminal state has been reached
        # end the episode and reset the environment
```

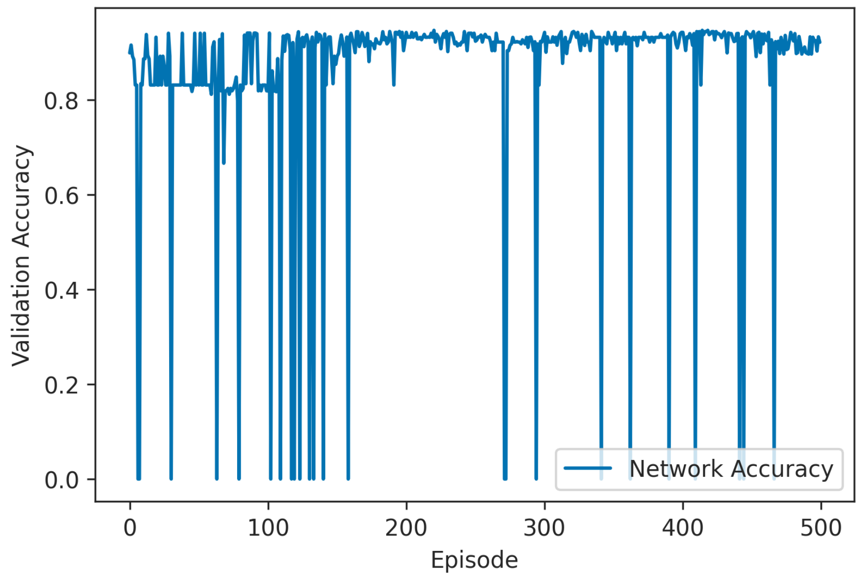


Figure 3.6: Network validation accuracy for 500 episodes of the reinforcement learning algorithm.

```
            if done:
                break
        # Update the target network, copying all weights and biases
        if i_episode % TARGET_UPDATE == 0:
            target_net.load_state_dict(policy_net.state_dict())
```

Listing 3.9: Reinforcement learning for NASBench-101

Chapter 3 References

[1] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *7th International Conference on Learning Representations, ICLR 2019*, 6 2018. [Online]. Available: http://arxiv.org/abs/1806.09055

[2] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, 7 2019.

[3] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 7 2018. [Online]. Available: http://arxiv.org/abs/1707.07012

[4] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-efficient distributed deep learning: A comprehensive survey," *arXiv preprint arXiv:2003.06307*, 2020.

[5] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *arXiv preprint arXiv:1909.02453*, 2019.

[6] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," *arXiv preprint arXiv:2001.00326*, 2020.

[7] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *International Conference on Machine Learning*. PMLR, 2019, pp. 7105–7114.

[8] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," *arXiv preprint arXiv:2008.09777*, 2020.

[9] A. Paszke, 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, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[10] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[11] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[12] G. Kyriakides and K. Margaritis, "Evolving graph convolutional networks for neural architecture search," *Neural Computing and Applications*, vol. 34, no. 2, pp. 899–909, 2022.

[13] P. Casale, O. Pujol, and P. Radeva, "Personalization and user verification in wearable systems using biometric walking patterns," *Personal and Ubiquitous Computing*, vol. 16, no. 5, pp. 563–580, 2012.

[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 11 2016. [Online]. Available: http://arxiv.org/abs/1611.01578

Chapter 4

Relative Ranking Retention Between Optimizers

As mentioned in earlier chapters of this thesis, the main barrier to entry for neural architecture search is the computational cost required to conduct experimental projects. Although most research focuses on improving existing methodologies and making them more efficient, the computational cost of evaluating neural architecture remains, even when such methods become indeed more efficient. This chapter investigates the ability of neural optimizers to retain relative rankings between neural architectures, utilizing an evolutionary algorithm to generate architectures and then training them with various optimizers

4.1 Introduction

A number of studies propose new NAS methods that can generate better networks or convert faster to the optimal architecture. Nonetheless, most neural architecture search papers are concerned with the behavior of the proposed method. Furthermore, these proposed methods spend most of their computational time evaluating intermediate solutions. As such, many of these methods employ distributed computational methods to speed up the process. Although this solution is practical and effective, it does not favor all families of algorithms, and requires additional hardware.

On the other hand, the networks are evaluated only to further the search and act as an intermediate step to generate the final architecture. As such, the absolute performance of these intermediate network solutions is not important. Their relative performance, on the other hand, is critical. Methods that can enable quick relative quality evaluation between architectures can significantly speedup a NAS search.

This chapter studies the utilization of various neural network optimizers under a reduced number of training epochs. We employ Monte-Carlo simulations, searching for optimal parameters in 3-d functions with added noise. The noisy and original function domains have a correlation coefficient similar to this observed in the reduced epoch training experiments. With this study, we aim to examine the extent of applicability for reduced epoch training as a proxy candidate evaluation task

4.2 Methodology

4.2.1 Evolving Architectures

Our methodology draws inspiration from CoDeepNEAT [1] to generate the neural architectures. However, we employ only the mutation operator, contrary to the original CoDeepNEAT, which employs speciated crossover and mutations. Our goal is to produce several diverse architectures of increasing complexity and not necessarily combine the best available architectures.

Architectures are initialized as a graph containing only two nodes; an input and an output. The mutation operator adds a new connection between existing nodes or a new node replacing an existing connection. In our implementation, nodes consist of the convolutional block proposed in [1]. Each block contains a convolutional layer with K filters and a kernel of size N, followed by a dropout layer with dropout probability d, a weight scaling factor S. A probability p exists of adding a max-pooling layer at the end. Concerning the number of connections and modules in the network, there are no algorithmic restrictions, although the networks were generated and verified (not evaluated) on a cluster of computers with NVIDIA GT730 cards. This imposed a practical upper limit on their sizes.

Our methodology makes use of genomes, chromosomes, and genes. A genome represents an individual, while a chromosome encodes information regarding its layers or connectivity. As such, each genome contains one connection chromosome and one block chromosome. Genes comprise a chromosome, and each gene contains information regarding a single connection (source and destination blocks) or a single convolutional block (K, N, d, S, p).

Some of the architectures generated during our experiments can be seen in Figure 4.1. The initial two-node architecture is depicted by number 1. Since NORD scales layers, the number of layers in a compiled network is significantly greater than the number of nodes in an architecture. For example, architecture 20 has 15 nodes corresponding to 86 PyTorch layers. Note that after the convolutional part of the network, a simple 2-layer dense head is added. Figure 4.2 shows the genome corresponding to architecture 6.

4.2.2 Experimental Setup

Our experiments consist of generating several unique architectures by evolving a single network for 20 generations. The networks were verified in a cluster of computers with NVIDIA GT730 GPUs and then sent to a cluster equipped with NVIDIA TESLA K40 for evaluation on the CIFAR10 dataset [2]. We train each compiled network for 1, 5, 10, 20, and 50 epochs, utilizing seven different optimizers: Adadelta, Adam, Adamax, RMSprop, Stochastic Gradient Descent (SGD), SGD with momentum (SGD-M) and SGD with Nesterov momentum (SGD-NM). Table 4.1 depicts the parameters of our setup.

To compare the ability of specific optimizer-epochs setups to act as accelerated proxy tasks, we want to investigate their ranking-retention qualities. Therefore, we first create lists of network rankings based on their test accuracy for each optimizer-epoch pair, resulting in 35 lists. Then, to compare the rankings under various setups, we employ Kendall's rank correlation coefficient τ [3]. Kendall's τ measures similarity between ranks (correlation of ranks), calculated utilizing concordant and discordant pairs. High positive values indicate high similarity

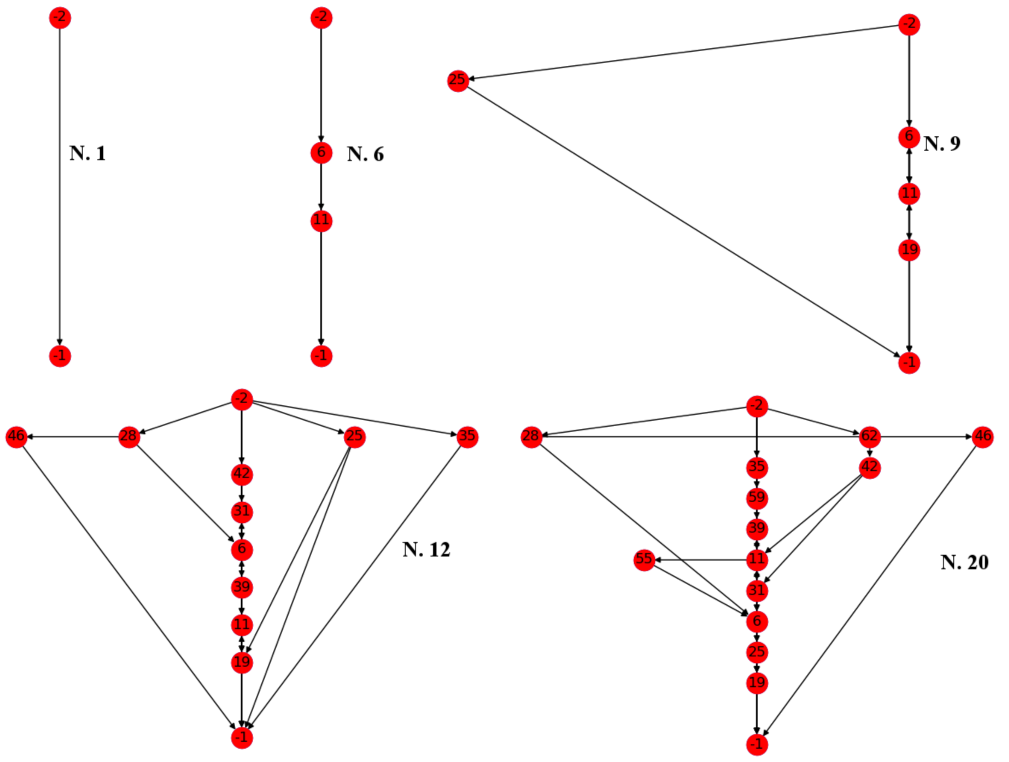


Figure 4.1: Sample architectures evolved.

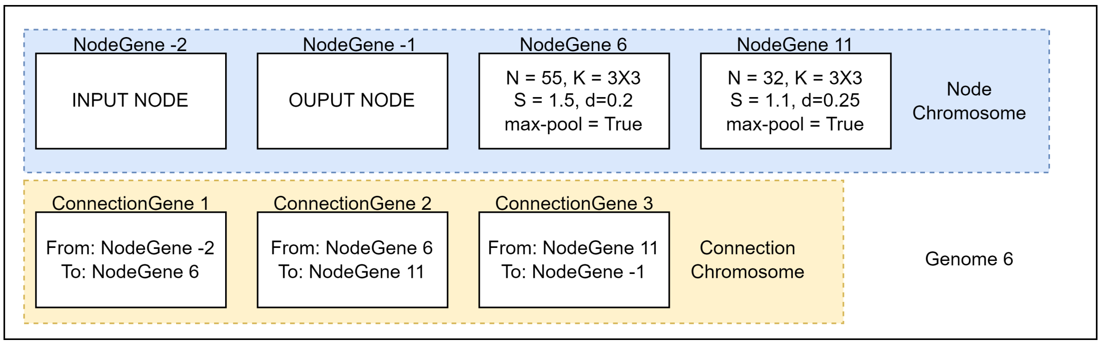


Figure 4.2: Genome example.

Table 4.1: Experimental Parameters.

Parameter	Symbol In-Text	Value
Number of Filters	N	[32,56]
Kernel Size	K	1, 3
Dropout Rate	d	[0, 0.7]
Scaling Factor	S	[0, 2.0]
Max-Pool Probability	p	0.5
Node Addition Probability	-	0.05
Connection Addition Probability	-	0.1
Generations	-	20
Epochs Trained	-	1,5,10,20,50
Number of Optimizers	-	7

between the rankings.

Given that Kendall's τ treats low-ranking and high-ranking pairs equally, we measure τ two times, one for the rankings between all architectures and one for the ten youngest architectures. As relatively simple architectures are easier to train, minor differences in initial conditions can lead to enough final accuracy differences to alter their ranks. On the other hand, more complex architectures require more effort to train successfully and, as such, are less likely to benefit significantly from favorable initial conditions.

After calculating the (Kendall) correlations between the various setups, we would like to explore what is their practical implication. For this reason, we conduct Monte-Carlo simulations utilizing a 3-d Rastrigin function. We choose the Rastrigin function, as it is multimodal, resembling the search spaces of neural architectures. First, we calculate the values for the function in a given domain. Given that the function is a mapping from a 3-d point to a scalar value, we can obtain rankings by re-arranging the 3-d points to a 1-d vector. Subsequently, we obtain a new mapping with a known Kendall correlation to the original by injecting pre-calculated random noise into the results. We fine-tune the noise to match the correlations observed in the optimizer-epochs setups.

Having an original and a noisy search space (original function's results and noisy results), we employ a simple genetic algorithm to find local minima in

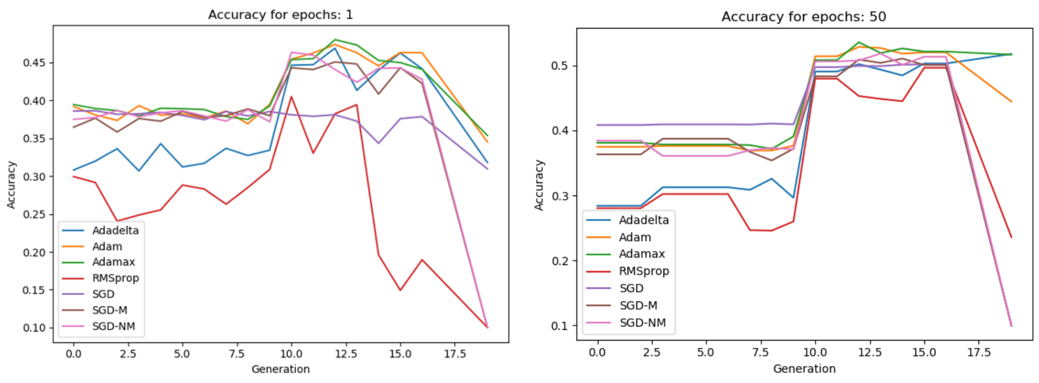


Figure 4.3: Comparison of achieved accuracy in 1 epoch of training to 50 epochs.

the two search spaces. First, we initialize a population of 10 individuals and implement a crossover operation with a probability of 0.9 and a mutation rate of 0.02. Parents are selected via tournament selection, and we experiment with 10, 100, and 1000 generations. Each experiment is repeated 1,000 times to obtain a reasonable empirical distribution.

4.3 Results

4.3.1 Neural Architectures

Although focusing on relative ranking under a reduced training epoch scheme for various optimizers, absolute accuracy values can hint at the overall quality of the networks. Figure 4.3 depicts the accuracy of each setup (1 and 50 epochs of training for the 7 optimizers). As is expected, 50 epochs provide the best results, stabilizing the rankings. Nonetheless, there seems to exist a pattern that both groups follow. There is a sharp increase in accuracy around generation 10 followed by relatively stable performance, and finally a noticeable drop in the last generations. Only the 'Ada' family (Adamax, Adadelta, Adam) seems to overcome the decrease in performance, given enough training epochs. The times required to train the networks are provided in Table 4.2. As expected, the average time required to train a network scales linearly with the number of epochs.

Table 4.2: Average training times (in seconds per architecture).

Optimizer	1 Epoch	5 Epochs	10 Epochs	20 Epochs	50 Epochs
Adadelta	36.1	158.6	312.3	622.1	1701.5
Adam	35.9	157.5	311.7	617.4	1724.8
Adamax	35.8	157.1	310.3	617.8	1718.6
RMSprop	35.4	156.5	310.2	614.6	1687.6
SGD	36.0	157.1	308.3	613.2	1687.3
SGD-M	35.7	157.2	309.3	613.5	1709.5
SGD-NM	35.5	156.6	311.2	617.7	1707.4

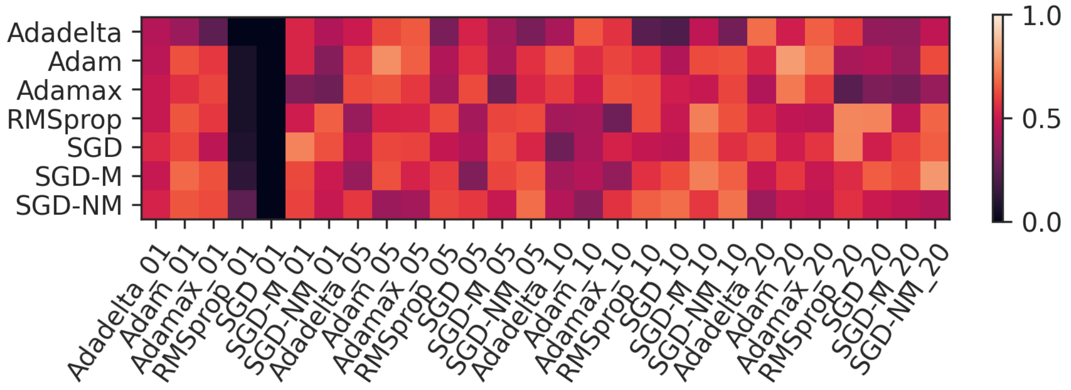


Figure 4.4: Rank correlation coefficients heatmap.

Judging relative performance from accuracy plots is not easy or accurate. To better understand the behavior of the architectures under different training setups, we compute the rank correlation coefficients for each ranking list. We regard the setups with 50 epochs as "fully trained" and compute Kendall's τ between all other lists and those generated by the 50 epochs setups.The results show high correlation coefficients (above 0.65) for most combinations, except for RMSprop_01 and SGD_01 as depicted in Figure 4.4.

Although promising and indicating a direction for future investigation, the results do not show clear or powerful correlations between the setups. This lack of strong correlation can be partly attributed to many simple architectures performing similarly and exchanging ranks due to favorable initial conditions. As such, we repeat the analysis by focusing on the 10 youngest architectures, which are significantly more complex, aiming to examine rank retention in such

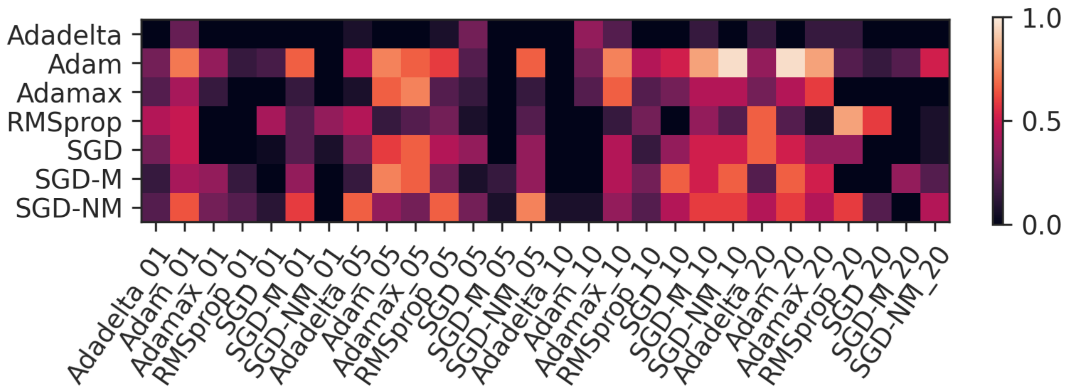


Figure 4.5: Rank correlation coefficients heatmap for the second half of generated architectures.

networks. As it can be seen in Figure 4.5, there are significantly stronger correlations between more complex architectures. Adam_01, Adam_05, Adam_20, Adamax_10, Adamax_20, RMSprop_20, SGD-M_10, and SGD-NM_10 have high correlations with Adam ($\tau > 0.75$, p < 0.05). Furthermore, SGD-NM_10 and Adam_20 have $\tau = 0.964$ with p < 0.01, indicating a strong correlation in relative rankings.

4.3.2 Monte Carlo Simulations

We have already indicated the need to accelerate NAS methods by utilizing proxy tasks. Furthermore, we have studied the rank correlations between various proxy tasks, obtaining significantly high results. Nonetheless, we do not yet know how such correlation levels impact an optimization method. The need to study this impact is even more significant in global search spaces, as architectures generated are more probable to exhibit differences similar to those in our current experiments.

To study such behaviors, we require a controlled environment where we have complete information regarding the correlations between the proxy and target tasks. This experiment enables the study of this behavior and, as such, the viability of using proxy spaces with specific correlation levels to the target. We pro-

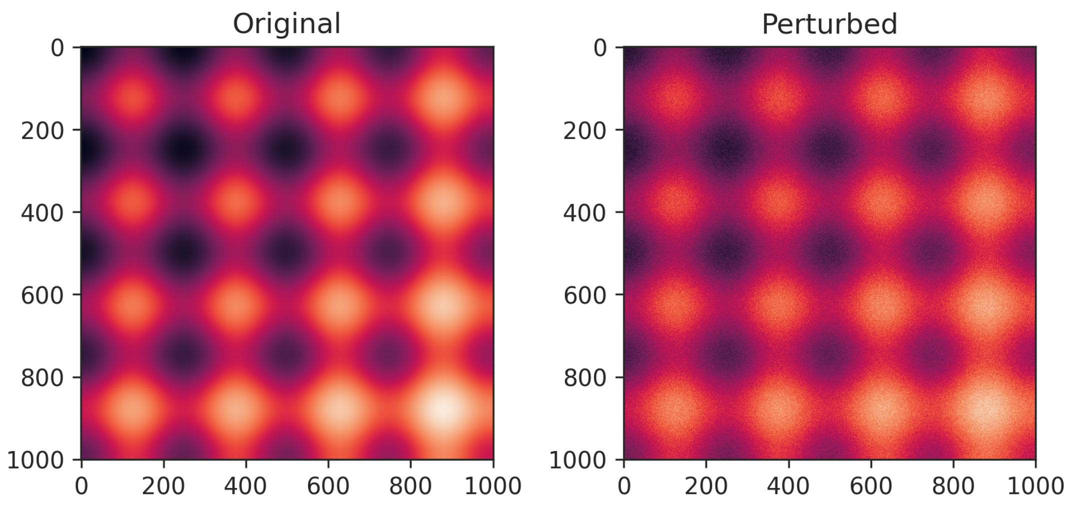


Figure 4.6: 2-D Rastrigin functions, original(left) and perturbed(right) with $\tau = 0.738$

ceed by generating the results for a 3-d Rastrigin function and injecting random noise into them until the required correlation to the original results is reached. A 2-d example of the two functions can be seen in Figure 4.6, which shows how the addition of noise changes the function's surface but leaves significant features and patterns intact.

We follow the same direction for the optimization method as in our original experiment by employing a genetic algorithm. The algorithm will search for solutions and utilize as a fitness function the noisy results. For the best solution found, we calculate the corresponding original function value and save the top % of solutions better than it (i.e. if a solution is better than 98% of all possible solutions, we save value of 2%) and repeat the process 1,000 times.

Table 4.3 depicts the top % average (E) while Table 4.4 the top % standard deviation of the generated solutions, over 1,000 runs. In the original space, the algorithm was able to generate solutions very close to the top-1%. For correlation levels of 0.75, the algorithm produced solutions near the top-4%, while for correlation levels of 0.85 and 0.95, it was able to produce solutions in the top-3% and top-2% respectively.

In a worst-case scenario ($\tau = 0.75$), the noisy function space produces so-

Table 4.3: Mean of achieved solution's top percentage for the original and perturbed search space, when applied to the original function.

Generations	E(original)	E($\tau = 0.95$)	E($\tau = 0.85$)	E($\tau = 0.75$)
10	0.160	0.162	0.184	0.195
100	0.050	0.058	0.086	0.103
1000	0.013	0.022	0.032	0.040

Table 4.4: Standard deviation of achieved solution's top percentage for the original and perturbed search space, when applied to the original function.

Generations	E(original)	E($\sigma = 0.95$)	E($\sigma = 0.85$)	E($\sigma = 0.75$)
10	0.160	0.162	0.184	0.195
100	0.050	0.058	0.086	0.103
1000	0.013	0.022	0.032	0.040

lutions in the range of [1.6%-8%]. For ($\tau = 0.85$) the solutions are within the [0.6%, 7%] range, while for ($\tau = 0.95$) they lie within the [0.3%, 3.9%] range. The solutions are marginally better in the original search space ([0%, 3.3%]), although the algorithm can find the global minimum. Moreover, we can see that as correlation increases, standard deviation decreases. Nonetheless, it is evident that the most important factor determining the final network's quality is the number of generations the algorithm is allowed to evolve.

4.4 Limitations

This chapter provides some interesting results, both from the real-world CIFAR-10 experiments and the simulations. Nonetheless, more thorough research is needed to determine the feasibility of reduced epoch training in more complex networks and cell search spaces. The architectures studied in this chapter are relatively limited. In the following chapter, we further our investigation, including more diverse architectures as well as cell search spaces.

4.5 Summary

This chapter studies the relative ranking retention of 20 distinct neural architectures when trained under various setups. The architectures were generated utilizing DeepNEAT's mutation operator. They were all trained using seven different optimizers for 1, 5, 10, 20, and 50 epochs. By comparing the relative ranking lists generated by each evaluation setup, we observed high correlations for most setups, except for RMSprop and SGD, when only a single training epoch is used. The more complex architectures exhibited higher ranking correlations compared to simpler architectures. By isolating them and computing rank correlation coefficients, a strong (τ= 0.964), statistically significant (p $<$ 0.01)correlation was revealed between the fully trained Adam optimizer group and SGD-NM when trained for 10 epochs and Adam when trained for 20.

To test the effect on search algorithms when searching on proxy tasks, we pre-calculate results for a 3-d Rastrigin function and inject noise into them, thus generating a "proxy" space for the Rastrigin function with a known correlation coefficient (0.75, 0.85, and 0.95). We then employ a genetic algorithm to optimize the function and save the percentage of all existing solutions better (on the original function) than the one generated. As a result, the algorithm was able to generate solutions belonging, on average, to the top 3,5% of the original function's solutions.

We observe that there is merit in using proxy search spaces, given that the computational resources freed from lengthy network evaluations can be reallocated to evaluate more networks. Absolute performance is not relevant when evaluating intermediate solutions. Instead, their relative performance is of paramount importance. Moreover, genetic algorithms seem to perofrm relatively well in noisy spaces, given that noise levels allow a positive correlation of at least 0.75 to the original space.Nonetheless, more thorough research must be conducted to establish optimal proxies for neural architecture design and optimization in various domains. In the following chapters, we will extend our experiments to larger datasets and utilize reduced-training distributed methods to generate state-of-the-art models for various problems.

Chapter 4 References

[1] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing.* Elsevier, 2019, pp. 293–312.

[2] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[3] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

Chapter 5

Training Candidates on Proxy Training Epochs

This chapter studies the effect of reduced training in neural architecture search. Building upon the results of Chapter 4, our experiments focus on relative rankings between architectures trained under various setups. We generate more diverse architectures and utilize better hardware, to further our investigation of ranking retention. Furthermore, we expand the investigation into cell-search spaces, specifically the search space of NASBench-101 [1]. Finally, we once again perform Monte-Carlo experiments, utilizing noisy spaces generated from Rastrigin functions.

5.1 Introduction and Relevant Works

In the previous chapter, we produced positive, although limited, results concerning the utilization of reduced training epochs as a NAS proxy task. The architectures and search space studied were quite limited. Moreover, new methods proposing more complex architectures continue to utilize proxy tasks. As such, we aim to thoroughly evaluate the impact of such evaluation schemes on a broader range of search spaces and architectures.

Most researchers utilizing this reduced-training method choose to fully train the top N produced models after the search phase. This fine-tuning of architectures significantly boosts network performance. One of the earliest works to perform reduced training was [2]. The authors evaluated architectures with only 8 training epochs while fine-tuning the best architectures with 120 training epochs. They report that utilizing few training epochs during search seems to produce architecture requiring far fewer training epochs to achieve performance comparable to the state-of-the-art. Deep multitask networks [3] also utilize a reduced evaluation epoch scheme to generate multi-alphabet character, recognition models. The authors utilize 3,000 iterations (forward and backward passes), while the 50 best models are trained for 30,000. Regularized evolution [4] during search also uses a small fraction (25 epochs) of the full training, 600 epochs.

The original NAS paper utilized 50 training epochs during the search phase while the final architectures are trained until convergence. On the other hand, the NASNet paper [5] utilized a mere 20 epochs while also training final architectures until convergence.

Even in ProxylessNAS [6] where a proxyless approach is proposed, employing an over-parameterized network with N possible paths, reduced epochs in the search phase are employed. Specifically, 200 epochs are used for the search phase, while the final network is trained for 600 epochs. Although producing satisfactory results, the method is computationally intense. It is logical that evaluating the actual deployment conditions will produce the best results.

Finally, Progressive Neural Architecture Search [7] utilizes a sequential model-based optimization approach and reduced search phase evaluation epochs. Here only 20 epochs are used, while the best models are trained for 300 epochs, and the best of these architectures is extracted as the final result.

From this brief overview of various NAS methods, utilizing reduced training epochs for the search phase is a widespread practice, aiding in its acceleration.Table 5.1 depicts the evaluation schemes of various methods, sorted by the ratio of final to search phase training epochs. The effect of employing such proxy tasks has been studied in the previous chapter. In this chapter, we explore more in-depth the effect and viability of using reduced training as a speedup method in NAS.

Table 5.1: Search phase and final training epochs of various methods.

Method	Search Phase	Final	Ratio
ProxylessNAS	200	600	3
Multitask CoDeepNEAT	3,000 (iterations)	30,000 (iterations)	10
CoDeepNEAT	8	120	15
Regularized Evolution	25	600	24
Progressive NAS	20	600	30

5.2 Methodology

Building upon Chapter 4, we generate 140 random, diverse architectures. First, we evaluate them on the CIFAR-10 dataset, expanding the understanding of relative ranking retention between more diverse architectures. Following, we utilize the NASBench-101 dataset [1], aiming to achieve two objectives. First, to demonstrate the noise injected by re-evaluating a set of architectures with different initial conditions. Second, to study the collective behavior of closely-related architectures under reduced epoch evaluation schemes. NASBench-101 has a relatively strict search space (only normal cells with at most 5 hidden layers and 9 connections). Finally, we once again perform Monte-Carlo simulations based on the correlation coefficients obtained from the first two experiments.

5.2.1 Generating Architectures

To generate several unique architectures, we build upon chapter 4. Again, we use blocks consisting of convolutional layers with K filters and kernels of size N, followed by a dropout layer with dropout probability d, a weight scaling factor S and possibly a max-pooling layer. The block architecture is depicted in Figure 5.1.

Each gene's parameters are sampled from a specific range, depicted in Table 5.2. Starting from the same simple architecture (only input and output nodes), we mutate the architecture by either perturbing parameter values, adding a block or a connection, or even removing a connection. Any invalid architecture is discarded, for example, if there is no path between the input and output nodes.

Table 5.2: DeepNEAT hyperparameters ranges used.

Hyperparameter	Range
Number of Filters (N)	[32, 256]
Dropout Rate (d)	[0, 0.7]
Weight Scaling (S)	[0, 2.0]
Kernel Size (K)	1, 3
Max Pooling (M)	True, False

In this chapter, we generate and validate the networks on NVIDIA TESLA V100 and K40 cards, significantly increasing the maximum parameter size for each network.

Figure 5.2 depicts some of the generated architectures. Note that contrary to chapter 4, some nodes may be orphaned (have no input or output connections, indicated with light cyan in the figure) as the mutation operator can now disable connections. Cells labeled "-2" indicate the architecture's input, while "-1" indicates the architecture's output (which is the input to the dense, 2-layer part of the network). Again, NORD scales layer outputs so they can be combined when a node has multiple inputs. In these architectures, following a Maximum Viable Size (as opposed to NORD's Minimum Viable Size) policy, leads to an explosion of inner network dimensions.

We generate a total of 140 distinct architectures and evaluate them on the CIFAR-10 dataset [8]. An average architecture has 70,000 trainable parameters, but all of them are in [9,099 - 148,140]. Although most networks do not have a high number of parameters, we focus our efforts on studying their relative performance under diverse training setups. We want to generalize our findings to novel tasks with unknown computational complexities. Therefore, it is more important to have a basket of diverse networks rather than a few over-parameterized. Each architecture is trained for 50 epochs(referred to as the fully-trained group, or FTG), as well as 1, 5, 10, and 20 epochs (referred to as partially-trained groups, or PTGs).

Moreover, for each version, we again employ the seven distinct optimizer setups; Adadelta, Adam, Adamax, RMSprop, Stochastic Gradient Descent (SGD),

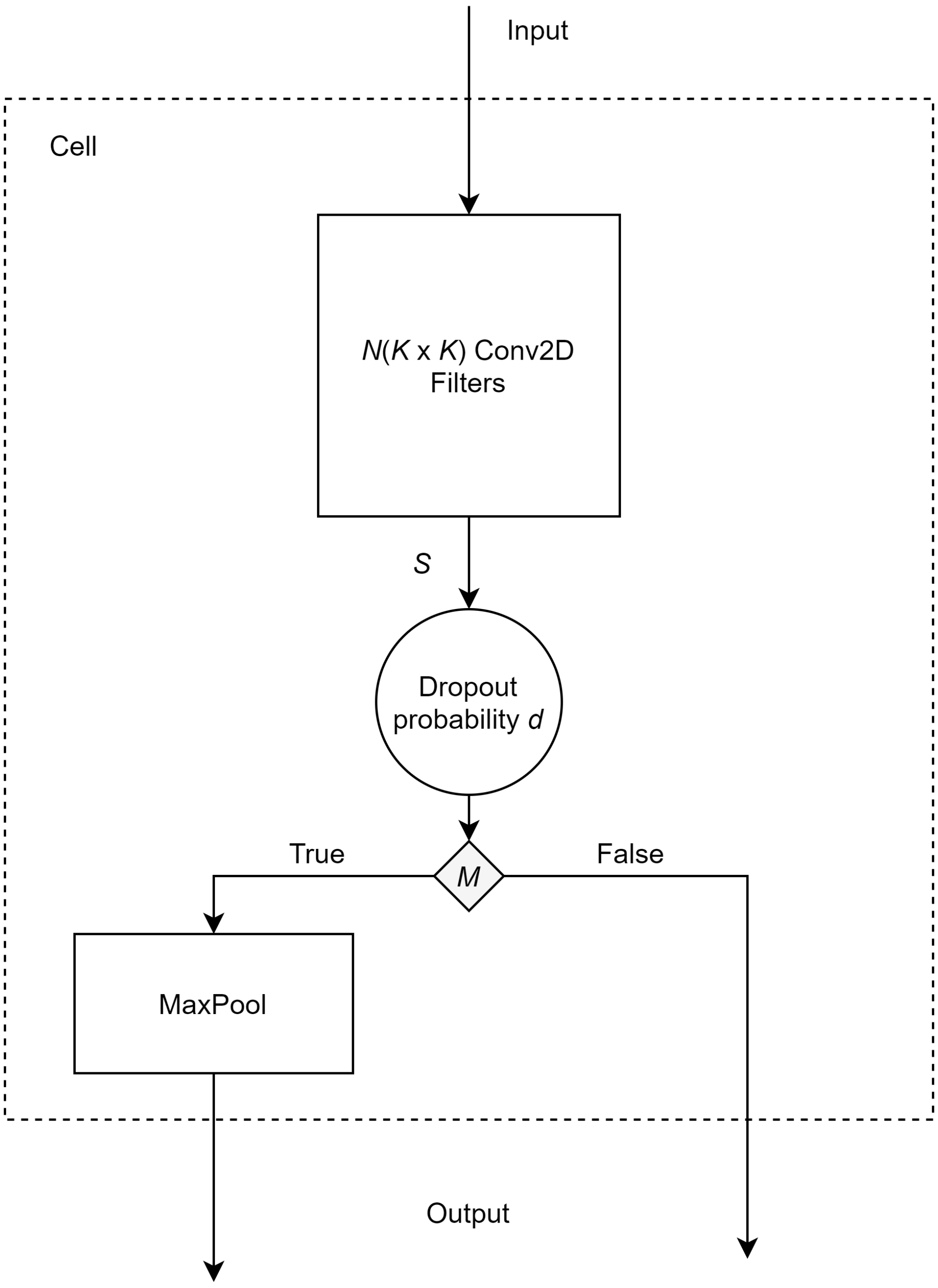


Figure 5.1: DeepNEAT cell architecture.

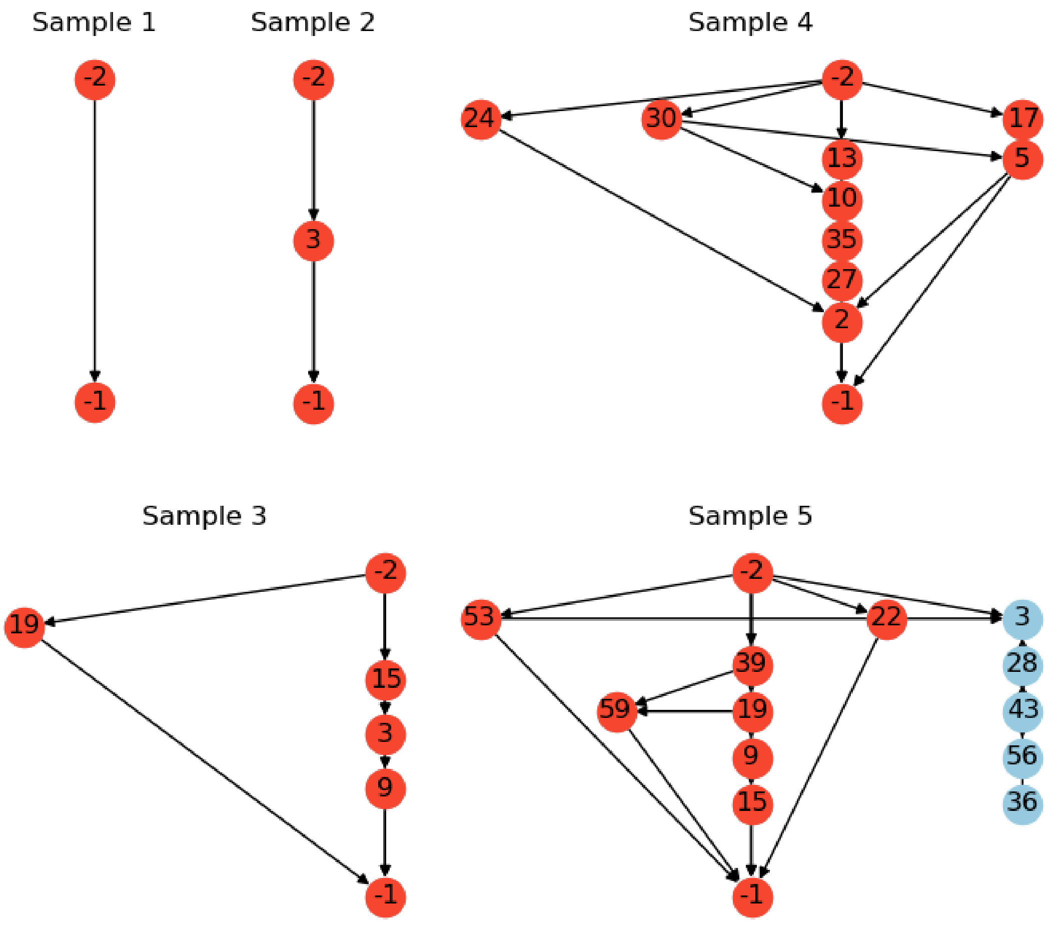


Figure 5.2: Example of neural architectures generated with the mutation operator. Light cyan nodes (Sample 5) are inactive.

SGD with momentum (SGD-M), and SGD with Nesterov momentum (SGD-NM). Again, calculating ranking lists for each setup and comparing the rankings, we employ Kendall's τ [9]. As our sample size is still relatively small (although significantly bigger than chapter 4), we generate 800 bootstrap samples to estimate τ [10]. Given that we rank each architecture by its test set accuracy, minor accuracy variations can result in significant variations between ranks. We saw an indication of this phenomenon in chapter 4, when we discarded the first half of the architectures and τ values increased significantly. As such, we calculate τ for the original accuracy and after rounding the accuracy to the closest integer percentage. Rounding eliminates significant ranking differences due to minor accuracy discrepancies, possibly induced by initial conditions.

5.2.2 NASBench-101 Simulations

To further study the behavior of networks evaluated under a reduced epoch scheme, we repeat the experiment utilizing the NASBench-101 dataset. Consisting of 423,624 cell architectures, it provides a sufficient number of samples. The outer skeleton, dictating how the cells are compiled into networks, is depicted in Figure 5.3). As inter-cell connectivity is pre-determined, generated architectures are relatively similar, contrary to architectures generated with unbound connectivity of global search spaces.

In this chapter's experiments, we sample 100,000 architectures with replacements. This sampling scheme aims to allow a real-world evaluation of architecture rankings. Each time an architecture is sampled results from one of three different training runs are retrieved. Our experiments compare results for 4, 12, 36, and 108 training epochs (PTG) to a specific 108 training run sample (FTG). We utilize the partially trained group of 108 epochs (108PTG) and FTG groups as a noise evaluator. Ideally, these two groups should have a Kendall's τ value of 1 (perfect ranking retention). Nonetheless, due to inherent stochasticity in the training process, we expect rankings to differ to some degree. Therefore, we aim to measure this discrepancy and establish a practical baseline, allowing us to fairly judge relative ranking retention between PTG and FTG.

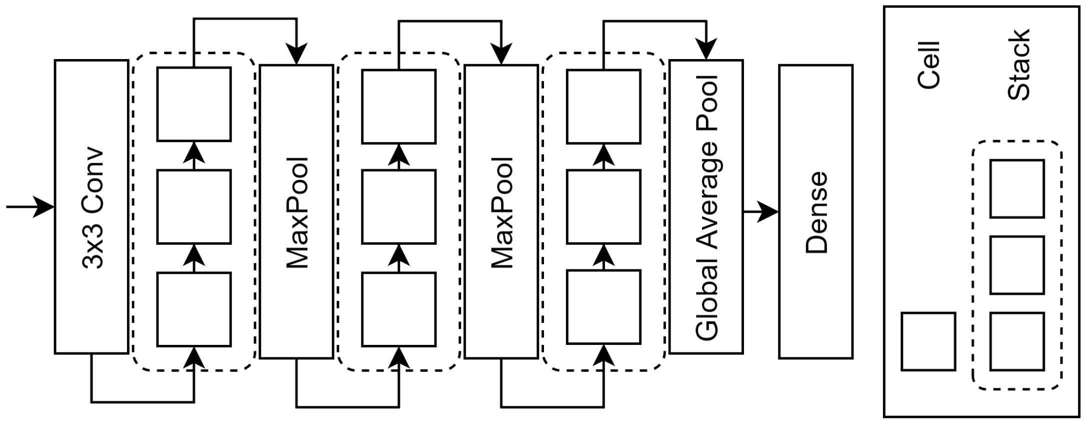


Figure 5.3: NASBench-101 fixed outer skeleton.

5.2.3 Monte-Carlo Simulations

As a final experiment, we aim to examine the ability of a genetic algorithm to find optimal solutions in a noisy environment that exhibit similar characteristics to those observed in the previous experiments. To simulate neural architectures' search spaces, we again utilize 3-d Rastrigin functions to demonstrate that optimization algorithms can find satisfactory solutions by exploring a noisy, albeit less computationally expensive environment. These simulated environments will have the same correlation to the original environment as the levels observed in this chapter's previous experiments. We record the percentage of better solutions in the search space for each solution generated. Although we could utilize NASBench-101 to conduct this experiment, it is restricted to a specific family of neural architectures, which introduces selection bias to the results. Rastrigin is a more general approach to evaluating optimization algorithms, and thus, a noisy Rastrigin function should translate better to a general NAS setting.

The genetic algorithm is employed to search both the noisy and the original search spaces. We generate noisy search spaces with correlations of $\tau = 0.85, \tau = 0.75, \tau = 0.65$. We follow the same procedure as chapter 4. First, we compute the function's values for all the original search space points. Following, we inject noise into the results to achieve the desired correlation level. To further expand the scope of our experiments, we also conduct experiments in 2 and 4-dimensional

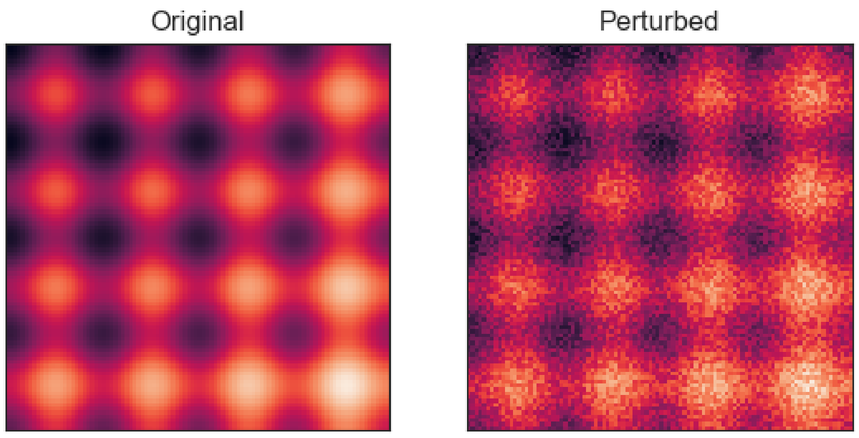


Figure 5.4: Rastrigin function 2D, 100 x 100 mesh grid. Lighter shades indicate higher values.

spaces, as well as the original 3-dimensional space. Figure 5.4 shows a 100-point 2-dimensional function and its noisy transformation. PyGMO [11], European Space Agency's general optimization library is utilized for the genetic algorithm implementation. A crossover probability of 0.9, mutation probability of 0.02, and a population size of 10 individuals are utilized, while tournament selection determines an offspring's parents. The algorithm's stopping criterion is a set number of generations, specifically 10, 100, and 1000. Finally, each experiment is repeated 100 times to create a sufficient sample of empirical results.

5.3 Results

In the following section, we present the results obtained from the experiments discussed in section 5.2.

Table 5.3: Average DeepNEAT accuracy for each optimizer for the fully and partially-trained versions.

Optimizer	1 Epoch	5 Epochs	10 Epochs	20 Epochs	50 Epochs
Adadelta	33.51	37.07	39.08	39.60	39.81
Adam	38.87	41.78	42.28	42.35	42.90
Adamax	38.84	42.02	42.62	42.44	42.45
RMSprop	29.55	31.05	32.66	33.16	33.79
SGD	35.41	38.82	39.32	40.48	41.40
SGD-M	38.30	41.88	43.07	42.23	43.65
SGD-NM	39.39	42.33	43.36	43.67	44.03

5.3.1 Generated Architectures

In the CIFAR-10 architectures experiment, we focus on relative ranking retention between architectures generated in a global search space. As Figure 5.5 shows, the accuracy distributions of PTGs tend to FTG when the number of training epochs increases. There seem to be at least three sub-populations in the sample, indicated by the three modes in the distribution. The right-most population first appears in the 5 epochs group (5PTG). Architectures that exhibit the highest performance separate from the rest, with the first 5 epochs of training contributing the most to this separation. Table 5.3 depicts the mean test accuracy for each training setup. Note that these statistics are calculated from the raw accuracies, not the rounded approximation mentioned in section 5.2.1. Mean accuracies show that networks are under-performing, which is expected as the networks are randomly generated. Parameters of successive layers and inter-layer connectivity are not designed to improve performance. Instead, this experiment is designed to exhibit the relative ranking retention under various training setups when network quality is distributed over a large area.

In Figure 5.6, a heatmap of the bootstrap-estimated Kendall's τ is provided between PTGs and FTG. We observe various PTG and FTG pairs with correlation values of $\tau > 0.5$, while Stochastic Gradient Descent and Adamax exhibit the highest correlations to PTGs, approximately $\tau \approx 0.6$.

Although there is a positive correlation, ranking again suffers from noise; small variations in final accuracy can induce significant variations in the rankings. As

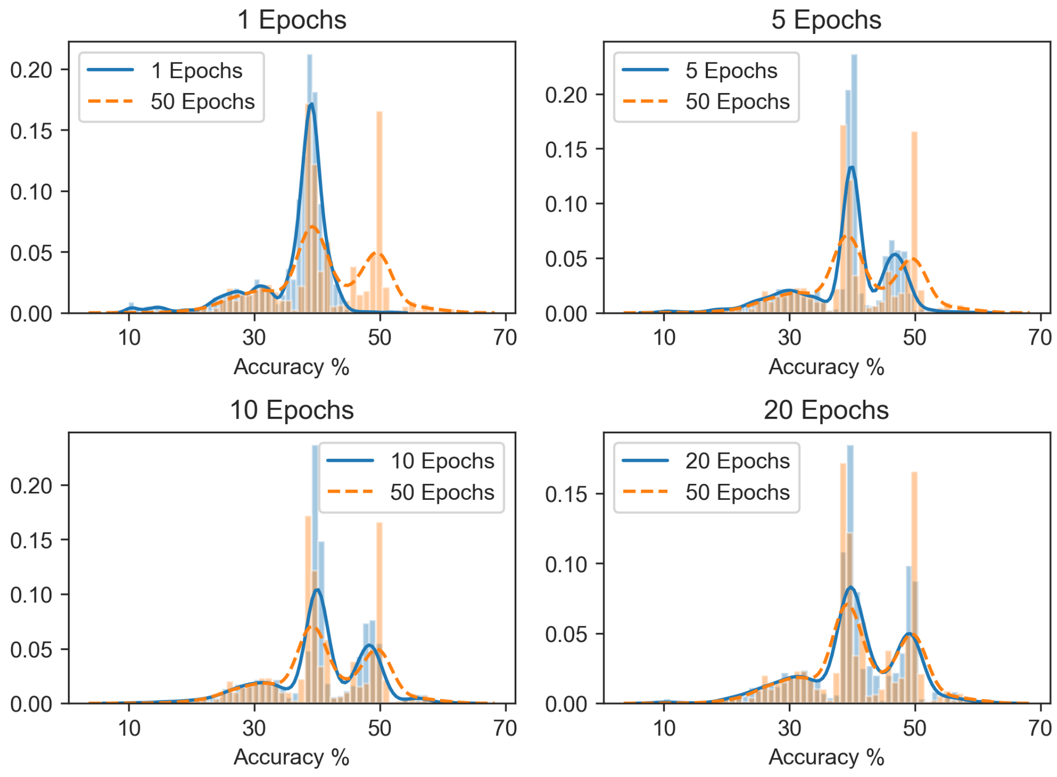


Figure 5.5: DeepNEAT accuracy distribution for 1, 5, 10, and 20 epochs, compared to 50 epochs.

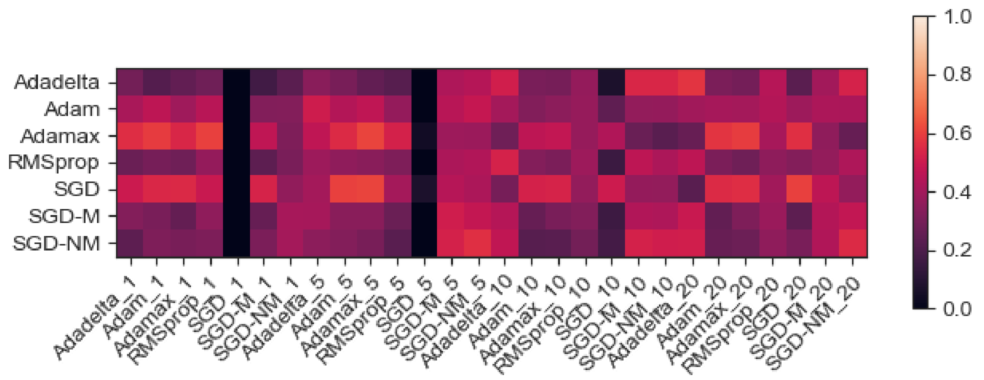


Figure 5.6: Bootstrap estimated correlations between fully-trained (y-axis) and partially-trained (x-axis) versions.

such, we re-evaluate Kendall's τ after rounding the accuracies to the nearest integer percentage. Results improve significantly (Figure 5.7), as Stochastic Gradient Descent and Adamax achieve correlations of $\tau > 0.7$. Moreover, FTG Stochastic Gradient Descent and 20PTG Stochastic Gradient Descent are correlated by $\tau = 0.851 \pm 0.028$. Figure 5.8 depicts bootstrap-estimated standard error. Pairs with a high correlation coefficient also seem to exhibit low standard error. Finally, as training epochs number increases, we observe that the average correlation between PTGs and FTG also increases. This increase is logical, as networks tend to converge to their full potential as the number of training epochs increases.

Results are promising, as FTG Stochastic Gradient Descent and Adamax are highly correlated with most PTGs, thus providing suitable optimizers for proxy training tasks. Furthermore, PTG Adam and Adamax show a consistently high correlation with FTG Stochastic Gradient Descent and Adamax. As such, they can safely be utilized as optimizers for the search phase of NAS.

5.3.2 NASBench-101 Results

In our NASBench-101 experiments, it is interesting first to examine the general behavior of various architectures as the number of training epochs increases. The dataset does not provide data regarding different optimizers, but it does provide data regarding various training epochs. As such, we provide Kendall's τ values

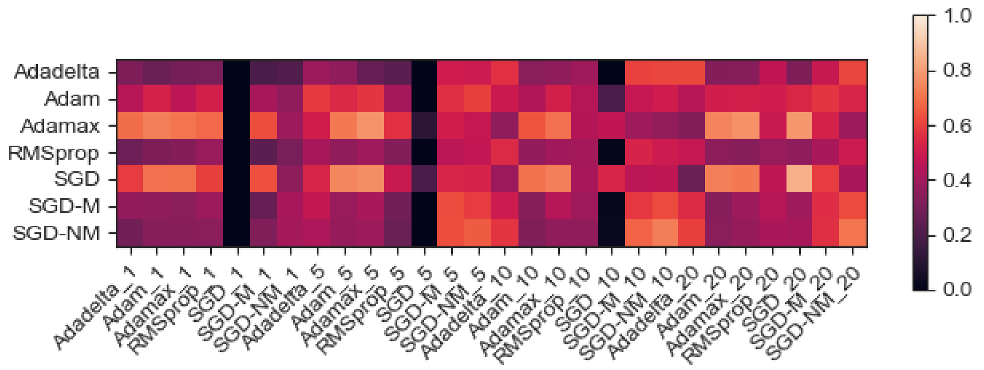


Figure 5.7: Bootstrap estimated correlations between fully-trained (y-axis) and partially-trained (x-axis) versions for *rounded* accuracies.

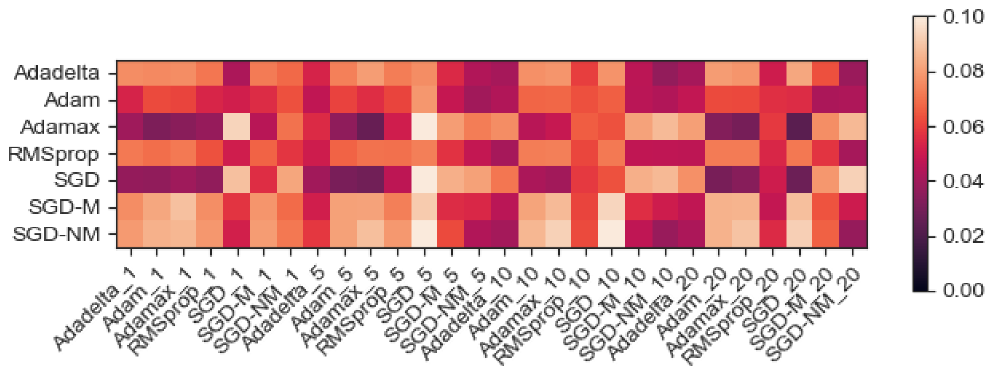


Figure 5.8: Bootstrap estimated correlation standard error between fully-trained (y-axis) and partially-trained (x-axis) versions for *rounded* accuracies.

for PTGs and FTG for the single optimizer employed in 5.9. Although 36PTG and FTG ranking correlation is 0.76, FTG and 108PTG (re-sampling of the 108 epochs accuracies) is only 0.89. This is far from the perfect correlation we would expect.

Furthermore, a 300% in available computational resources resulted in a 17% improvement in ranking correlation. The comparison of 108PTG to FTG shows the consequences of re-training a group of architectures with the same training setup but different initial conditions and inherent stochasticity in the training process (for example, out-of-order execution in GPGPU instructions). The impact is significant, inducing a ranking correlation significantly lower than expected. Note that there is a 33% probability of re-sampling the same accuracy value for every single network. Nonetheless, when a large number of networks is evaluated, the probability of sampling the same values twice is effectively zero. This results in the inability to achieve the theoretically expected correlation and imposes a practical upper limit on the expected correlation between proxy task and original search space.

For each partially trained group (PTG) and the fully trained group (FTG), we calculate the bootstrap-estimated τ after rounding their accuracy to the closest integer percentage. This results in a higher ranking retention for 36PTG and 108PTG. Figure 5.11 depicts the bootstrap distributions of the estimated values and the standard error. By applying the Lilliefors test [12] we conclude that the values are normally distributed, allowing us to use the standard error to construct confidence intervals. As Figure 5.11 depicts, 108PTG achieves on average a ranking correlation of 94%, a considerable improvement over the non-rounded rankings (Figure 5.10). Although re-training the same architectures induced noise, rounding reduced its effect considerably.

Moreover, we observe improvements in the correlations for 36PTG, as it produces, on average, rankings 80% similar to FTG. These results are close to those observed for Adamax and SGD in the previous experiment, although NASBench employs RMSProp with cosine learning rate decay. Nonetheless, a correlation of 80% is feasible in both search spaces. Partially trained groups with 4 and 12 epochs of training exhibit low levels of correlation with FTG. More train-

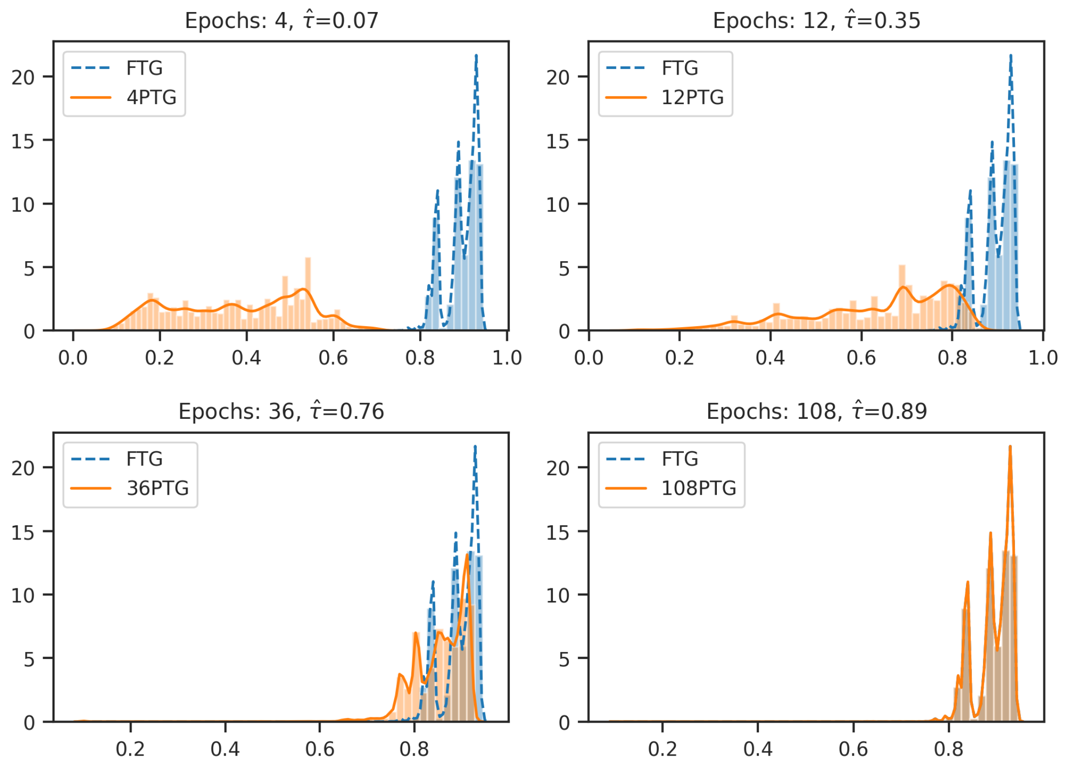


Figure 5.9: Accuracy distributions and Kendall's tau-b between PTGs and FTG for NASBench.

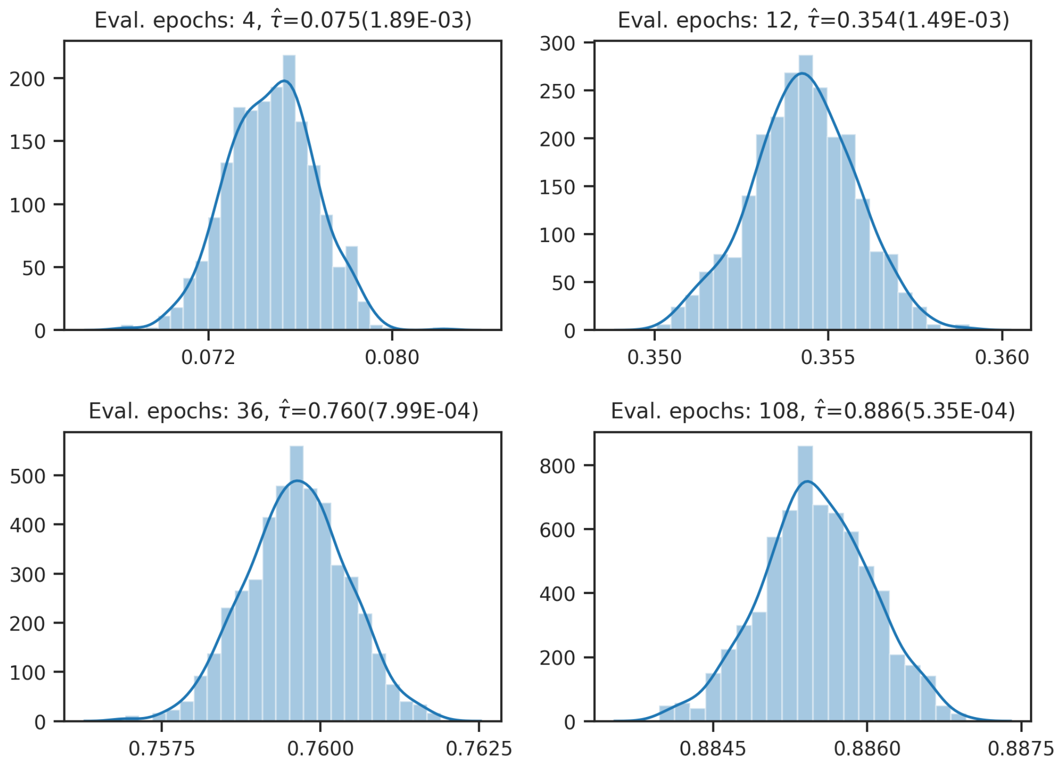


Figure 5.10: Bootstrap estimated distributions for Kendall's tau-b on the *rounded* NASBench dataset. The number in brackets denotes the standard error.

ing epochs were needed to achieve similar correlation levels to the global search space experiment. This overhead can be attributed to the nature of the search space, which leads to generating architectures with similar characteristics. As final networks share a large portion of their macro-architecture, more minute details discern between average and exceptional architectures. Consequently, more training epochs are needed to highlight those differences.

The results of this experiment further validate the viability of utilizing reduced training epochs to speed up the search phase of NAS. Furthermore, it highlights the need for an increased number of epochs when generated architectures share major features and the difficulty in achieving high correlation coefficients even when architectures remain the same between training groups.

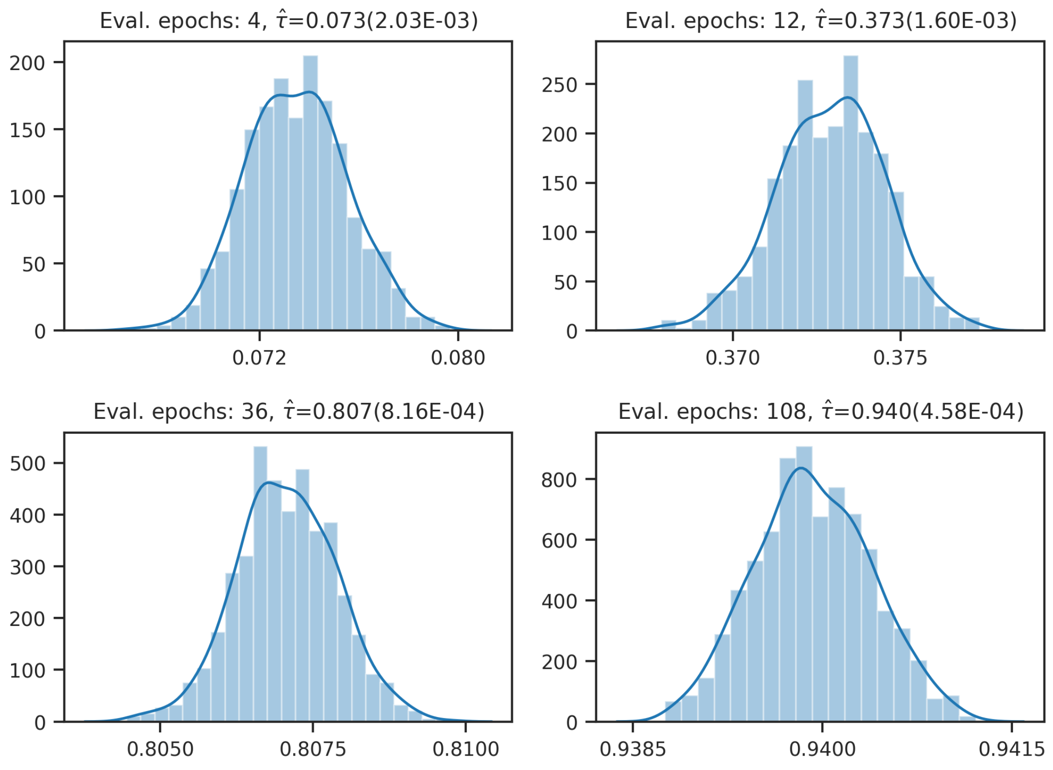


Figure 5.11: Bootstrap estimated distributions for Kendall's tau-b on the *rounded* NASBench dataset. The number in brackets denotes the standard error.

5.3.3 Monte-Carlo Results

For our last experiment, we search for optimal solutions to a noisy Rastrigin function. Employing a genetic algorithm, we search for spaces with τ levels in {0.65, 0.75, and 0.85} (65SP, 75SP, and 85SP, respectively). Correlation levels are chosen to represent a worst, average, and best-case scenario, given the results of the previous two experiments. Table 5.4 depicts the results of the simulations. Given the global search space experiment's results, we can assume that a proxy task with a correlation level of $\tau = 0.75$ to the original search space (OSP) translates to a speedup of ± 10 (Figure 5.6, Adamax to Adamax_5 correlation is 0.779). We first compare the performance of a 10-generation search on the OSP and a 100-generation on 75SP (roughly the same computational cost). We observe that, on average, the 100-generation 75SP run produces better results (quality measured on the original search space). This behavior also applies when performing a 100-generation search on the OSP and a 1000-generation search on 75SP. A 95% confidence interval shows that for the 1000-generation run, 75SP was able to generate solutions in the top [2%, 7.5%].

However, when employing 1000-generation runs, the benefits of utilizing a proxy search space with $\tau = 0.75$ seem to decrease as the number of dimensions increases. This can be attributed to the fact that the number of neighboring points increases for a given point in space as dimensions increase. When we inject noise to the values of each point, we create clusters of similarly-valued solutions, which increase the difficulty of distinguishing between marginally better solutions. This behavior is similar to ranking correlations under various training setups observed in the previous two experiments. It seems to validate our findings further, indicating more significant gains from utilizing reduced training in global search spaces, where solutions are more diverse and spread out in the search space. Contrary to this, cell-search spaces benefit less from reduced epoch training, as most of the architectures cluster around the space generated by the fixed macro-architecture.

For 85SP, the genetic algorithm can find even better solutions, although the expected speedup (assuming the computational behavior observed in the first

Table 5.4: Expected performance when searching in the original and perturbed search spaces. Expressed in the top-N percentage of the original search space (smaller is better).

Dimensions	Generations	E(original)	E(τ=0.85)	E(τ=0.75)	E(τ=0.65)
	10	0.131	0.141	0.180	0.196
2	100	0.042	0.072	0.085	0.111
	1000	0.002	0.032	0.033	0.044
	10	0.161	0.179	0.199	0.217
3	100	0.043	0.076	0.091	0.127
	1000	0.002	0.030	0.038	0.046
	10	0.177	0.195	0.220	0.243
4	100	0.044	0.084	0.098	0.133
	1000	0.002	0.033	0.043	0.048

experiment) is insufficient to justify its usage. Nonetheless, the merit of utilizing 1000-generation searches on the 85SP compared to 100-generation searches on OSP is prevalent even in the 4-dimensional version. For fewer generations, its usage is not justified, as 75SP proves to be more efficient (in terms of solution quality to computational cost). Finally, 65SP and 100-generation runs, solutions discovered outperformed 10-generation runs on OSP. Assuming the speedup observed in the global search space experiment (e.x. Adamax to Adamax_01 offering a speedup of 50), even the 1000-generation runs can be effectively utilized, as it provides considerable improvements over 10-generation runs on OSP.

From the above, we conclude that when choosing between the increase of the allocated resources of the NAS optimization algorithm (in our case, the number of generations) or the resources of intermediate solution evaluation (training epochs), the former is a safer choice. For severely restricted search spaces, where differences between realized networks are minute, increasing the allocated resources (training epochs) of the evaluation is preferable.

5.4 Limitations

This chapter provides some interesting insight into the tradeoff between allocating resources to the search algorithm or the candidate evaluation method. Nonetheless, there are a number of limitations in our experiments. Although the number of architectures and their diversity increased significantly from chapter 4, the sample size remains small for the first experiment. Furthermore, we are still utilizing single-GPU evaluation for each architecture, thus limiting the architectures in size. Although having a significantly bigger sample size, the second experiment is constrained by the family of architectures in the dataset.

Finally, both datasets concern convolutional networks evaluated on the CIFAR-10 image recognition dataset. Therefore, there is a possibility that selection bias has skewed the results in the form of sampling bias. Following these limitations, if selection bias is indeed present, it also skews the results of the third experiment, as the correlation levels are set according to the findings of the first two experiments. In this case, results are only valid for sub-populations with characteristics similar to samples utilized in the first and second experiments.

5.5 Summary and Conclusions

In this chapter, we have studied the behavior of neural architectures when evaluated under a reduced training epoch scheme based on their relative rankings. To study this behavior, we created 140 unique networks utilizing mutations and trained each network for 1, 5, 10,20, and 50 epochs using seven different optimizers: Adadelta, Adam, Adamax, RMSprop, Stochastic Gradient Descent (SGD), SGD with momentum (SGD-M), and SGD with Nesterov momentum (SGD-NM). In addition, we applied a rounding operation to their test accuracy to reduce the noise generated by inherent stochasticity in the training process, rounding to the nearest integer percentage. Finally, the rounded accuracies were utilized to generate rankings lists for each experimental setup (epochs and optimizer variation). We compared the relative rankings for each reduced-epoch setup to the fully-trained setups, employing Kendall's tau ranking correlation coefficient.

Given that our sample size is relatively small, we generated 800 bootstrap samples to estimate the tau distributions. We observed high correlations between fully-trained SGD setups and the 20-epoch SGD setup (0.85 ± 0.028). We concluded that SGD and Adamax are suitable optimizers to utilize in training final NAS architectures, as they exhibit a high correlation with various partially trained optimizer setups. Furthermore, Adam and Adamax are suitable search phase optimizers, as they generated a high correlation coefficient with fully trained SGD and Adamax.

To further investigate the level of relative ranking retention between fully and partially trained networks, we studied the behavior of 100,000 architectures sampled from the NASBench-101 dataset. As the dataset does not provide data for different optimizers (only RMSProp is utilized), we tested for 4, 12, 36, and 108 epochs. This experiment provided more insight, yielding relatively high correlation coefficients ($0.808 \pm 8.33E - 4$) for the 36-epoch partially trained group (36PTG) and the fully-trained group (FTG). However, it also indicated that it is impossible to reproduce the same ranking between two groups of independently trained networks, even if the architectures are identical (108PTG and FTG). This inability can be attributed to inherent stochasticity in training the networks. Essentially, we cannot directly measure the quality of a neural architecture. We can only observe the quality of a compiled network derived from the architecture.

Even when rounding the accuracy to the nearest integer percentage, the correlation between 108PTG and FTG was only $0.94 \pm 4.72E - 4$, posing a practical upper limit on what can be achieved. Finally, as the standard deviation of the fully-trained performance distribution decreases, the number of epochs required to rank architectures increases.

To test the ability of optimization algorithms to find optimal architectures in a proxy search space, we generated noisy Rastrigin function results with rank correlation coefficient levels in 0.65, 0.75, 0.85 (compared to the original function results). We then employed a genetic algorithm to search for optimal solutions in the noisy search spaces. The algorithm produced solutions, on average, within the top-4% 5.4 of the original search space for $\tau = 0.75$. This experiment further validated previous findings, showing that in lower-dimensional spaces, and spaces able to generate diverse solutions, the algorithm was more efficient when extra computational resources were allocated to the search rather than candidate evaluation. As such, low-correlated spaces are beneficial. High-correlated spaces are required to generate optimal solutions in search spaces where solutions are clustered around a single point (such as one defined by a macro-architecture).

We conclude from the above that it is not detrimental to utilize a reduced number of training epochs during the NAS search phase. A highly accurate evaluation of candidates is not needed and may be detrimental as it reduces the available computational resources for the search method. Accurate analysis of their *relative* performance is sufficient to produce near-optimal solutions. Furthermore, even when fully re-training a set of neural networks, their rankings may differ significantly from what someone would expect.

There is a certain point where the tradeoff between a more intense candidate evaluation and a more thorough search becomes apparent, and it is directly related to the diversity of architecture generated. Fewer epochs are needed to discern between diverse architectures, while more epochs are needed for similar architectures.An extreme example of diverse architectures would be an architecture with only an input and an output node and an architecture with several convolutional layers. To rank them, even a single training batch is sufficient. Ont the other hand, two identical networks of several convolutional layers, differing only in the kernel size of a single layer by 1 pixel, will have very similar performance, and even a large number of training epochs may not be sufficient to rank them successfully.

This chapter has thus successfully informed us about the utilization of reduced epochs training as a candidate evaluation method. Adam and SGD seem to be safe optimizer choices. Furthermore, it highlighted the merit of allocating more computational time to the search method, rather than the candidate evaluation method, especially for global search spaces. Diverse architectures seem to differentiate in performance early on in the training phase, while more similar architectures require more training effort. In the following chapters, we utilize this knowledge to search for optimal architectures in image and graph datasets, utilizing distributed computing and reduced training epoch schemes.

Chapter 5 References

[1] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *International Conference on Machine Learning*. PMLR, 2019, pp. 7105–7114.

[2] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312.

[3] J. Liang, E. Meyerson, and R. Miikkulainen, "Evolutionary architecture search for deep multitask networks," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 466–473.

[4] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, 7 2019.

[5] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 7 2018. [Online]. Available: http://arxiv.org/abs/1707.07012

[6] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[7] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," pp. 19–34, 2018. [Online]. Available: http://github.com/tensorflow/

[8] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[9] M. G. Kendall, “A new measure of rank correlation,” *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[10] B. Efron and V. Petrosian, “Nonparametric methods for doubly truncated data,” *Journal of the American Statistical Association*, vol. 94, no. 447, pp. 824–834, 1999.

[11] F. Biscani and D. Izzo, “A parallel global multiobjective framework for optimization: pagmo,” *Journal of Open Source Software*, vol. 5, no. 53, p. 2338, 2020.

[12] H. W. Lilliefors, “On the kolmogorov-smirnov test for normality with mean and variance unknown,” *Journal of the American statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

Chapter 6

Distributed Regularized Evolution for Global Search Spaces

Most research focuses on searching for small blocks of deep learning operations or micro-search. This method yields satisfactory results but demands prior knowledge of the macro architecture's structure. Generally, methods that do not utilize macro structure knowledge perform worse but can be applied to datasets of entirely new domains.

This chapter proposes a macro NAS methodology that utilizes concepts of Regularized Evolution and Macro Neural Architecture Search (Deep-NEAT) [1] and applies it to the Fashion-MNIST dataset. By utilizing our method, we can produce networks that outperform other macro NAS methods on the dataset when the same post-search inference methods are used. Furthermore, we can achieve 94.46% test accuracy while requiring considerably fewer epochs to train our network fully.

6.1 Introduction

Most successful methodologies in NAS concern cell search spaces, which require significant prior knowledge about the target dataset. Although these approaches are both efficient and effective, the prior knowledge required to generate the macro architecture does not exist for all types of data sets. Therefore, they are only applicable in areas where such knowledge has been previously obtained which usually means thorough human experimentation. As mentioned previously in this thesis, experimentation requires significant computational resources and time. On the other hand, global search spaces require more computational resources than cell search spaces but can be implemented for data sets where previous knowledge regarding well-performing macro-architectures does not exist.

In this chapter, we design an approach created by combining regularized Evolution from [2] and genetic encoding from DeepNEAT [1], combined with the knowledge extracted from the previous chapters. Utilizing a global search space, we search for optimal architectures in the fashion-MNIST dataset[3], employing a reduced-training-epoch candidate evaluation method. We aim to evaluate the proposed method's ability to generate suitable architectures without prior knowledge regarding the application domain and compare it to other approaches with similar set spaces. We first briefly presented other approaches and the target data set. Finally, we explain our approach and present our experimental results. Limitations are discussed at the end.

6.2 Related Work

As previously mentioned, we utilize a combination of Regularized Evolution [2] and DeepNEAT [1] to formulate a global NAS methodology for our method. As such, in this section, we present the basics of the two algorithms and the Fashion-MNIST dataset.

6.2.1 DENSER

In DENSER [4], an evolutionary algorithm is employed to design neural architectures. The algorithm searches for the network's topology and optimal hyper-parameters, such as learning rate parameters, other optimizer parameters and data augmentation parameters. The search space consists of a two-level hierarchical search space. The lower level encodes parameters regarding individual layers, while the higher level encodes parameters regarding network connectivity.

Generating architectures involves utilizing context-free, human-readable grammar. The method is applied to a variety of image recognition datasets. For Fashion-MNIST, it is able to produce networks with 94.23% accuracy while employing 10 epochs in the search phase and 400 in the final training phase.

6.2.2 Auto-KERAS

In [5], network morphisms and bayesian optimization are utilized in order to generate neural architectures. Identity morphisms are utilized by inducing function-preserving transformations and allowing re-utilization of previously trained weights. Bayesian optimization is used to guide the morphisms, increasing their efficiency in altering architectures. Applied to the Fashion-MNIST dataset, utilizing 200 epochs for both search and final training, the method generated architectures with 93.28% accuracy.

6.2.3 DeepSwarm

In DeepSwarm, a population-based algorithm, specifically ant colony optimization (ACO), is employed to generate competent neural architectures. The generated networks consist of sequentially connected layers, which can be convolutional, pooling, and batch normalization layers. There are no skip connections or branching layer outputs. Nonetheless, utilizing 50 epochs for candidate evaluation and 100 for the final training, the method is able to out-perform [5], producing networks with 93.56% final accuracy.

6.2.4 NASH

Finally, a simple hill-climbing algorithm is utilized in NASH to design neural architectures. Again, network morphisms are employed, but instead of having a Bayesian optimization algorithm guide the insertion, a simple hill-climbing algorithm is employed. Here, skip connections exist and the basic operations that the hill-climbing algorithm can choose from are the following:

- Making the network deeper, by adding a convolutional block (convolution followed by a batch normalization and activation function).
- Making the network wider increasing the number of channels used by its internal layers.
- Adding a skip connection from a single layer to another.

6.2.5 Fashion-MNIST

Fashion-MNIST [3] is a benchmarking image recognition dataset. It was first proposed as a direct drop-in replacement for the popular MNIST dataset [6], which had become too easy for modern deep learning architectures to provide a benchmark. The dataset contains several labeled fashion items grouped into 10 classes. Each instance is a 28 by 28 pixel grayscale image depicting a single item. There are 60,000 items in the training set and 10,000 items in the test set. An example of the dataset can be seen in Figure 6.1.

6.3 Methodology

Building upon the knowledge gained from the previous chapters, this chapter's methods utilizes concepts of Regularized Evolution [2], DeepNEAT [1], as well as reduced epoch candidate evaluation methods. Following, we define the rules of evolution taken from [2]. Finally, we explain how a genome (a collection of genes) is translated into a functioning neural network and evaluated.

6.3.1 Architecture Representation

In order to represent a neural architecture, genes contain information regarding their layers and connections. These genes define the network's topology, as well as its layers parameters and layer type.

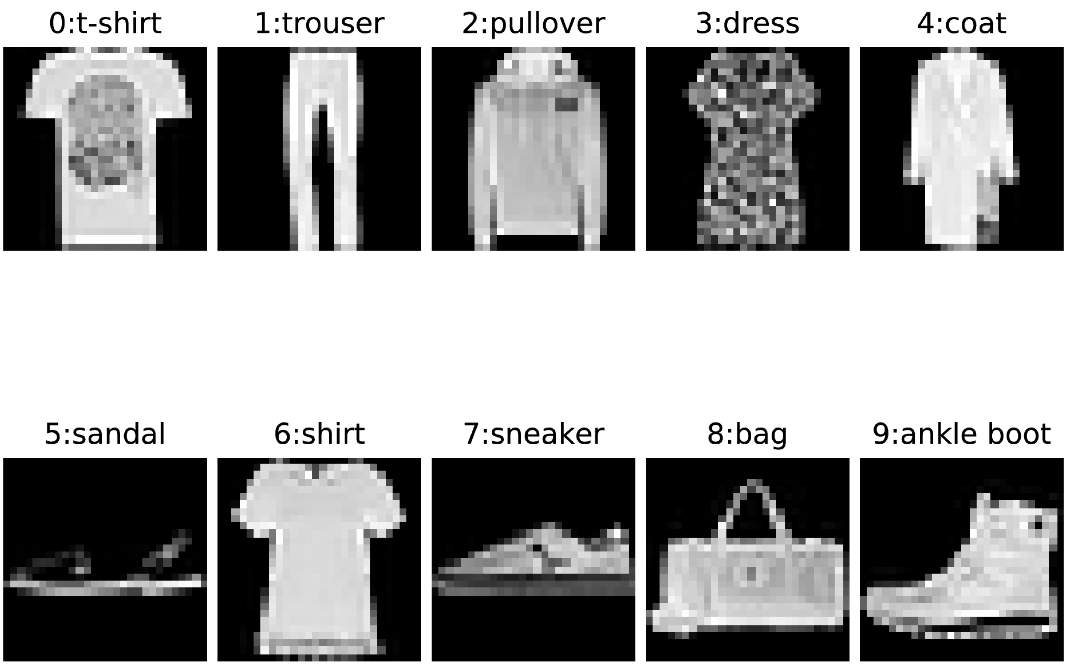


Figure 6.1: Sampled Fasion-MNIST images, one for each class.

Layer Genes

Each layer gene dictates which layer type will be implemented by a single node. Deviating from our previous approaches, we opt to implement 12 discrete layer configurations instead of having variable kernel size, number of filters, dropout probability, as well as the existence of the MaxPool layer. This approach dramatically reduces the search space, which is already vast, given that no connection restrictions apply.

To specify the layer configurations, we first define a fixed number of channels for all architectures, as in [2]. Each node can implement either a convolution or a pooling layer, with kernels of sizes 2x2, 3x3, and 5x5. Furthermore, each pooling layer may implement a max or average reduction function (being a Max-Pool or Average-Pool). Finally, each convolutional layer may choose to implement only half of the predefined channel (number of filters) number. This decision reduces the available variations in each node from 896 down to 12. The available choices

Table 6.1: Available layer choices.

Selection	Layer Type	Kernel Size	Operation
CONV_2.H		2x2	
CONV_3.H	Convolution	3x3	Filters Number = Half Channels
CONV_5.H		5x5	
CONV_2.N		2x2	
CONV_3.N	Convolution	3x3	Filters Number = All Channels
CONV_5.N		5x5	
POOL_2.A		2x2	
POOL_3.A	Pooling	3x3	Average
POOL_5.A		5x5	
POOL_2.M		2x2	
POOL_3.M	Pooling	3x3	Max
POOL_5.M		5x5	

are depicted in Table 6.1.

Connection Genes

Connection genes indicate the network's connectivity and are implemented in the same manner as previous chapters. The only difference is that mutation operators can only disable a connection and not generate a novel connection. Furthermore, this choice does not restrict the search space to sequential architectures, i.e., there can exist networks with two or more layers with the same inputs or a layer with multiple inputs. Although this may seem to restrict the search space, skip connections can still form, as it will become clear later.

6.3.2 Evolving Architectures

Following the method in [2], a population of P individuals is initialized. Each new individual represents a 4-layer architecture. Again, this does not restrict the existence of architectures with fewer than 4 layers in the population. As connections can be disabled, some layers may be orphaned and thus become inactive themselves. To evolve the population, we implement a mutation operator, which is applied with probability p_{mutate}. We further define a separate probability to

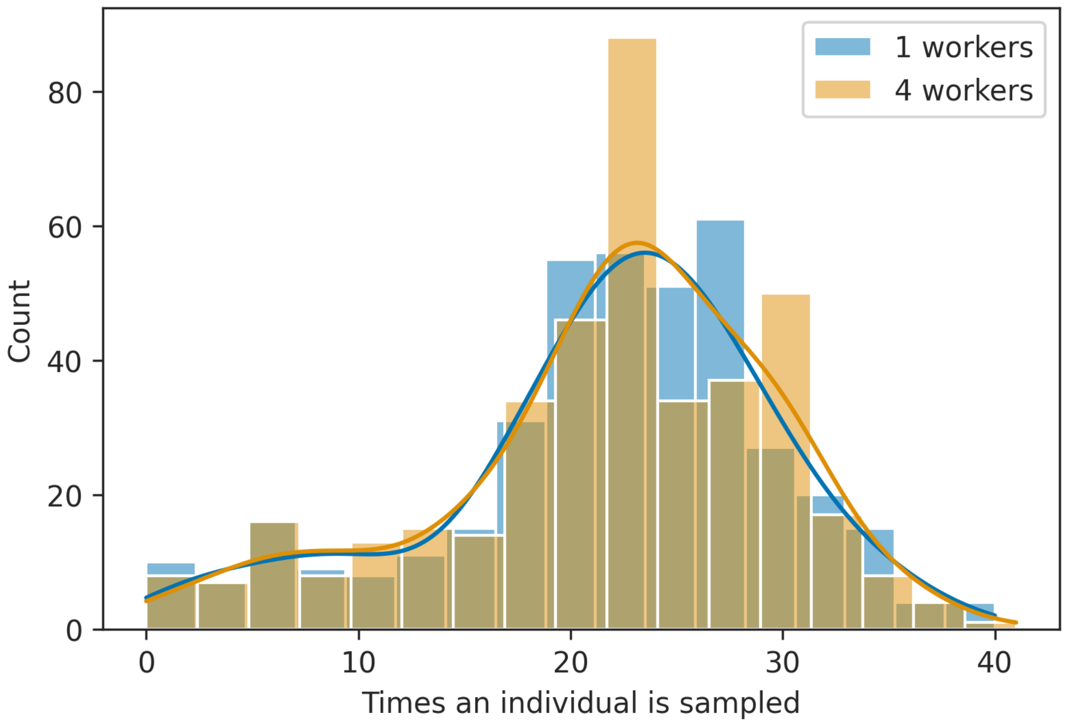


Figure 6.2: Times an individual is sampled for serial (1 worker) and distributed (4 workers) version of regularized evolution.

add a new node to the network p_{add}, as well as a probability $p_{identity}$ to leave the network intact. At each evolution cycle (a cycle consists of replacing the oldest individual with an offspring), N individuals are selected as candidates for reproduction. The best (utilizing the compiled neural network's test set accuracy as its fitness) out of N is selected to create offspring inserted into the population, and the oldest individual is discarded. Having W parallel GPU workers, we employ concurrent evolutionary cycles, where at the end of each cycle, W offspring are created, and W oldest individuals are discarded. This does not significantly alter the probabilities of an individual to be sampled. Figure 6.2 depicts the probability that any individual will be sampled a given number of times before being discarded, for serial and distributed implementations.

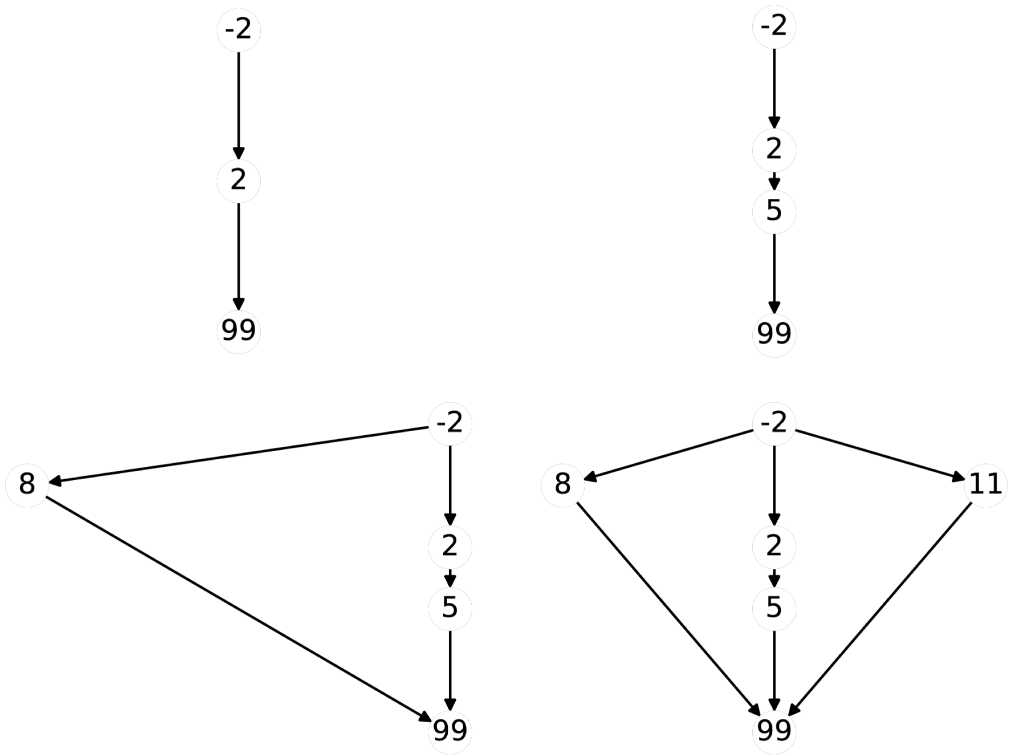


Figure 6.3: Three node additions to the same genome.

Adding Nodes

An existing connection is selected at random to insert a new node in an architecture. The new node is inserted and connected to the original connection's source and target node while the connection is disabled. Note that the connection is *not* discarded but instead is de-activated. When applied to an inactive connection gene, the mutation operator re-activates it, thus allowing for the creation of skip connections. With this mechanism, the original source node can have multiple outputs, as each time a new node is inserted, a direct connection to the original target is retained, although it is inactive. This can be seen in Figure 6.3. Each node is identified by a unique number, indicating its age. Node -2 is the network's input, while 99 is the network's output. As it can be seen, the input node has developed multiple outputs, while the output node has developed multiple inputs.

6.3.3 Implementing the Networks

To utilize the convolutional architectures for the classification of the Fashion-MNIST dataset, we need to define a dense head. We add three fully connected layers (FC) with a decreasing number of filters. The first fully connected layer (FC1) has a number of neurons equal to a quarter of its input features. The second fully connected layer (FC2) has half the number of neurons that FC1 has. Finally, the third fully connected layer has 10 neurons, equal to the number of classes in the dataset.

After each convolutional layer, a ReLU activation function is applied and clipped to have a maximum value of 6 (ReLU6). Following the activation, a Batch Normalization layer is employed, along with a dropout layer with dropout probability $d = 0.1$. Minimum Viable Size is employed by NORD to ensure that layers with multiple inputs can sum them before processing their inputs.

6.4 Experiments

6.4.1 Experimental Setup

In this chapter's experiments, we employ a population of 100 genomes. Utilizing a total of 400 evolution cycles, we evaluate a total of 500 networks. At each evolution cycle, we select 25 individuals as prospective parents. Each network during the search phase is trained for 10 epochs and evaluated. To train the networks, we utilize the Adam optimizer [7], which proved to be suitable for NAS search phases. We choose 64 as the fixed channel number. This means that CONV_2.H, CONV_3.H, and CONV_5.H will contain only 32 filters when selected as layer configurations. Furthermore, the probability of adding a new node to an architecture is set to $p_{add} = 0.25$, while the probability of adding a new connection is 0.7. As such, the probability of leaving a network intact is $p_{identity} = 0.05$.

To establish a baseline for our method, we also implement a random search approach. For this purpose, we repeat the experiment with $N = 1$, meaning that a random individual is selected each time to reproduce. This reduces the algorithm to a random search. The experiments were run on a cluster of 4 NVIDIA Tesla V100 GPU cards, employing our distributed algorithm implementation.

6.4.2 Results

Our distributed evolutionary search (ES) implementation seems to significantly outperform random search (RS). It was able to produce better architectures while doing so consistently. The results can be seen in Figure 6.4, where the currently best-found architecture accuracy is depicted for both approaches. ES manages to continuously generate better architectures, while RS cannot do the same (left). Moreover, the evolutionary search produces better empirical cumulative distributions, exerting a first-order stochastic dominance over random search. Stochastic dominance implies that ES can produce at least the same or better architectures with a higher probability. ES can generate architecture achieving test accuracy of up to 93.62%. This architecture consists of two CONV_2.N layers, followed by a CONV_3.N, a CONV_5.N, a POOL_5.A, and a CONV_3.H layers, all of them sequentially connected. RS was able to find architectures with up to 91.91% test accuracy, although these architectures consisted of a CONV_3.H, CONV_5.H, a CONV_3.H, and a POOL_5.M layer, again connected sequentially.

Most of the best-performing architectures found by ES consisted of sequentially-connected layers with full channel count, followed by a single pooling layer and a reduced-channel convolution. This can be attributed to the dense head selected when designing our search space. Some architectures developed diverging outputs and multiple inputs, achieving relatively good performance. The most prominent features were two sequential CONV_2.N layers, followed by a CONV_3.N, a CONV_5.N, and a POOL_5.A layer. Figure 6.5 shows all evolved architectures (initial population not included), each with an alpha equal to $\frac{1}{400}$. This allows robust features to be represented in bold while less prominent features appear extremely faded. This is an easy way to identify patterns visually. Here, OUTPUT layers denote the output of the convolutional part of the network, i.e., the input to the dense head. Valid evolved architectures contained, on average, 5.5

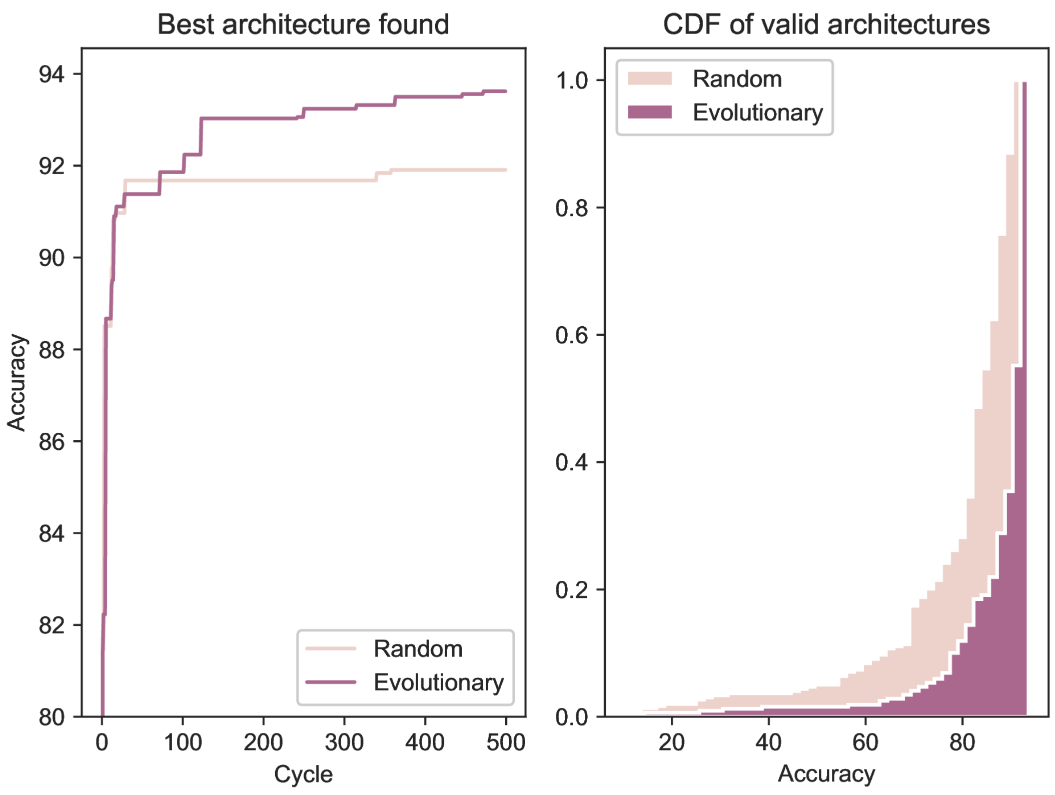


Figure 6.4: Best architecture's performance (left) and performance empirical cumulative distributions (right).

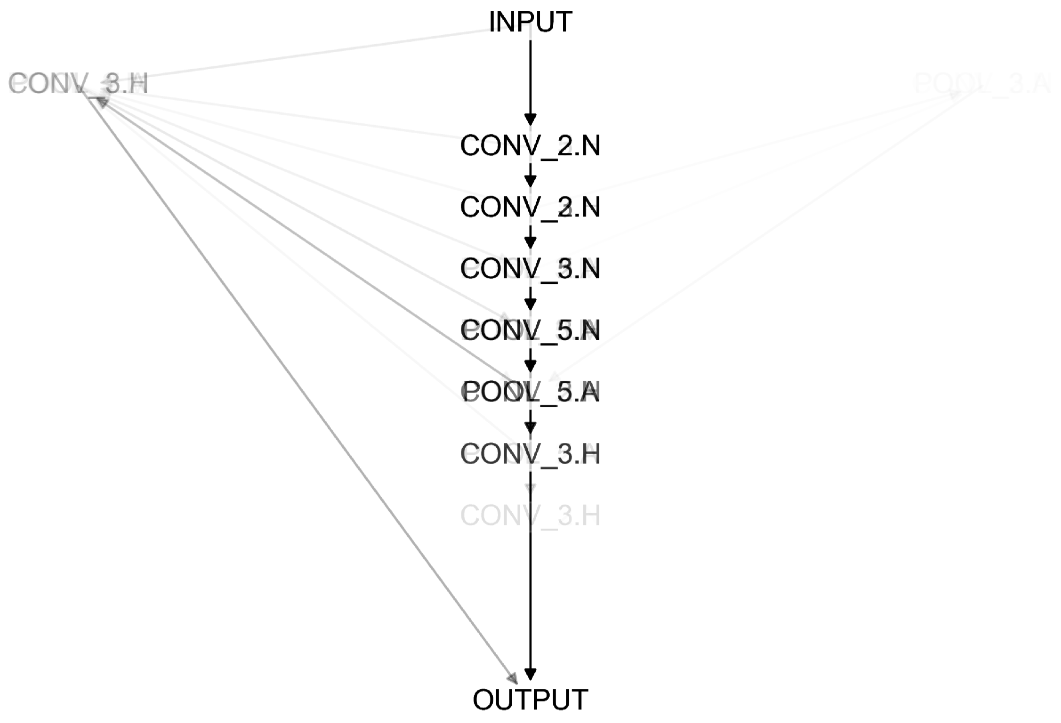


Figure 6.5: Overlay of all genomes' architectures.

nodes with a standard deviation of 1.1. Some invalid architectures were generated, including architectures with no path from the input to the output node. In Figure 6.5 a pattern of 4 convolutions of increasing kernel size is prominent, as is the branching of outputs from the main sequence to a convolution layer with kernel size 3, feeding the output layer. It is interesting to note that the dominant patterns seen in Figure 6.5 were evolved many times and were then established due to their performance.

We chose the best-performing architecture out of architectures trained at least 5 distinct times for the final training. We select the average best, as we want to refrain from choosing a network that achieved high performance due to stochasticity. This decision follows the work of [8], as similar levels of variance were recorded in this study (0.2% on average). The chosen network closely repre-

Table 6.2: Performance of similar algorithms

Method	Final Accuracy	Search Epochs	Training Epochs
DENSER [4][2]	*94.23%*	**10**	400
Auto-Keras [5]	93.28%	200	200
Byla2019 [9]	93.56%	50	100
NASH [10][3]	91.95%	20-105	205
REMNet-128	*94.26%*	**10**	**20**
REMNet-256	**94.46**%	**10**	**20**

sents the dominant architecture from Figure 6.5 (first 5 sequential layers shown, followed by two CONV_3.H layers branching out after the pooling operation). Employing data augmentation on the train set (horizontal flipping and random erasing), increasing the number of channels to 256 (resulting in 3.1 million parameters) and training the network for 20 epochs with Adamax as optimizer, after which it converged. The network can achieve a test set accuracy of 94.46%, while a more modest implementation with 128 channels (791,338 parameters) achieved a test accuracy of 94.26%. We call the architecture REMNet (Regularized Evolution for Macro-NAS Net) for ease of reference.

6.4.3 Discussion

Compared to networks found by other global search space NAS methodologies, our generated network seems to perform marginally better. Nonetheless, there exist post-search training and inference methods that can greatly boost network performance, such as test-set data augmentation and neural network ensembles used in [4]. By leveraging such methods, the authors are able to propose an architecture with 94.7% accuracy when test-set augmentation is used and 95.26% when ensembles are employed. As we do not employ any of these methods, we refrain from comparing our model with those versions.

Table 6.2 depicts a summary of the results obtained by our method and other similar approaches. REMNet seems to need fewer epochs to fully train the final

[2]94.70% with test set augmentation

[3]As implemented in [5]

network, which can be attributed to two distinct reasons. First, we employ a significantly restricted candidate evaluation scheme with only 10 epochs, as we are confident that it is enough to evaluate network performance for up to 50 epochs in the final training. This also imposes a regularization on the candidate selection, as overly complex networks are not able to distinguish themselves from well-performing simple networks. Second, we utilize optimizers with an adaptive learning rate. Thus, we do not need to employ complex learning rate policies, as in [4].

6.4.4 Limitations

Though our approach produced better results than similar methodologies, it cannot produce better than state-of-the-art architectures. This is partly due to the search space selection, as it does not allow the exploitation of known promising regions. Furthermore, we do not employ advanced training and evaluation techniques for the final architecture, thus being unable to unlock its full potential.

6.5 Summary

There are several methods that can produce state-of-the-art networks through cell search space NAS. Nonetheless, these methods require prior knowledge regarding suitable regions of the search space. In this chapter, we propose a methodology for global search spaces by combining Regularized Evolution [2], and DeepNEAT [1]elements while also utilizing knowledge gained from the previous chapters, employing reduced epoch training as a candidate evaluation method.

By applying our approach to the Fashion-MNIST data set, we were able to produce neural architectures that outperform other similar search space methodologies when the post-search training evaluation methods are similar. We designed networks capable of up to 94.46% accuracy on the test while converging in relatively few epochs.In the following chapter, we aim to leverage our method of generating relatively standardized (fixed layer configurations) neural architectures in order to create a model that can predict relative performance without requiring any training on networks compiled from the neural architectures.

Chapter 6 References

[1] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, “Evolving deep neural networks,” in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312.

[2] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized evolution for image classifier architecture search,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, 7 2019.

[3] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.

[4] F. Assunçao, N. Lourenço, P. Machado, and B. Ribeiro, “Denser: deep evolutionary network structured representation,” *Genetic Programming and Evolvable Machines*, vol. 20, no. 1, pp. 5–35, 2019.

[5] H. Jin, Q. Song, and X. Hu, “Auto-keras: An efficient neural architecture search system.” Association for Computing Machinery, 7 2019, pp. 1946–1956.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[8] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, “Hierarchical representations for efficient architecture search,” *6th International*

Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings, 11 2018. [Online]. Available: http://arxiv.org/abs/1711.00436

[9] E. Byla and W. Pang, "Deepswarm: Optimising convolutional neural networks using swarm intelligence," *UK Workshop on Computational Intelligence*, pp. 119–130, 2019. [Online]. Available: https://github.com/Pattio/DeepSwarm

[10] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

Chapter 7

Graph Neural Networks and Neural Architecture Search

In this chapter, the networks we generate (performance predictors) will learn how to rank other neural architectures based on their topologies. Again, we employ concepts from regularized evolution [1], and genetic algorithms [2] to generate a number of global search space architectures. We aim to utilize the generated architectures as an evaluation dataset and search for a Graph Convolutional Network that can correctly predict the relative performance of other architectures. This chapter addresses relative performance as a binary classification problem (which network is better). Although there have been successful attempts at employing graph convolutional networks to model the same domain, to the best of our knowledge, this is the first time that a neural architecture search methodology is employed to generate a neural predictor that aims to accelerate other NAS methods. Furthermore, this is the first time that evolutionary algorithms have been applied to the problem of NAS for GCN.

7.1 Introduction

So far, we have addressed the problem of increased computational requirements in NAS by exploring how to allocate available resources best. This chapter attempts to significantly reduce the requirements by making training compiled networks obsolete. Estimating an architecture's quality from a predictive model (as opposed to computationally observing it through the performance of a compiled network) can induce a significant speedup in a NAS pipeline. There is an added cost of training the model, although the benefits can potentially outweigh the cost [3, 4, 5]. Having a suitable surrogate model can also aid in Bayesian Optimization approaches [6, 7, 8]. Both of these methods induce unavoidable noise to the NAS process and affect relative architecture rankings. We showed in chapter 5 that even retraining networks compiled from the same architectures could induce noise. As noise is unavoidable, researchers must find the optimal trade-off between speedup and precision for their specific application, as it is impossible to achieve perfect precision.

In this chapter, we aim to propose a NAS method that can create suitable predictive models for other NAS methods, thus significantly automating the process of choosing one. First, we present related works. Following, we present the basic components of our approach, experiments and results. Finally, we discuss the limitations of our approach and our findings.

7.2 Related Works

7.2.1 Predicting neural network performance

Most NAS approaches that do not employ differentiable operations require the evaluation of compiled networks. These evaluations have been identified as the main computational burden of NAS, requiring resources that could be allocated to the search algorithm. As a result, some researchers opted for predictive models instead of utilizing a reduced epoch evaluation scheme seen in previous chapters. BRP-NAS[9] employs a four-layer GCN to predict binary rankings between various architectures. This approach presents some similarities to our approach, presented later in this chapter. Nonetheless, the authors focus on predicting inference latency, and the predictive model is derived from a hand-crafted architecture. The authors also employ transfer learning to transfer the network optimized on the latency prediction problem to the binary ranking problem and produce satisfactory results for both problems.

In [4], the researchers employ an auto-encoder model to create representations of each neural architecture, which are then used to organize all architecture representations in a graph. Radial basis functions are employed to measure the distance between two architecture representations. The calculated distance is then used to position the architecture in a final relational graph. This supergraph is then utilized as an input to a graph convolutional network model to assess the performance of various architectures (represented as nodes in the supergraph). Though satisfactory results were produced, the need for two separate models and a supergraph can be seen as drawbacks.

ReNAS[5] is a more straightforward approach to the task of assessing relative performance between cell search spaces architectures. Here, the cell's adjacency matrix and layer types are employed as a feature tensor (similar in spirit to the reinforcement learning example from chapter 3) and an extended version including the number of parameters for each layer and floating operations per second. A modified version of the LeNet-5 convolutional architecture [10] is employed as the prediction model. The network is trained on either the 19x7x7 full feature vector or the 1x7x7 reduced vector (adjacency and layer type only). This method has produced state-of-the-art results, outperforming all previous methods which utilized LSTM embeddings[11] and Random Forests[12]. Furthermore, it seems to outperform [4] for train sets with an equal number of samples.

7.2.2 Neural architecture search on graph neural networks

As NAS can produce state-of-the-art architectures for convolutional networks, it is only logical that other types of neural networks can benefit from its methods. Graph convolutional networks are no exception.

GraphNAS [13] employs a reinforcement learning approach to designing variable-length strings representing hyperparameters of graph neural networks (GNNs). Networks are evaluated on a number of classification tasks and protein-protein interaction graphs. The researchers employ a policy gradient approach, using the network's accuracy on each task as the reward. Moreover, the authors employ weight-sharing between each generated architecture to reduce the search's overall computational and time costs.

Similarly, Auto-GNN [14] also employs a reinforcement learning and weight sharing approach to design graph convolutional architectures for the same datasets. Additionally, the authors employ a guided architecture modifier, aiming to assist the agent in exploring the action space. This guided modifier leverages the decision entropy of various actions to determine how an existing neural architecture should be modified. This approach outperforms GraphNAS on the selected datasets.

Finally, SNAG [15] introduces a novel framework for generating GNN architectures. Similar to the previous two approaches, this method employs NAS with reinforcement learning. Nonetheless, it utilizes a more simplistic and expressive search space. Here, the search space includes 12 node aggregators, 3 layer aggregators, an Identity, and a zeroize layer. Additionally, the agent is optimized to maximize the expected validation accuracy of the generated architectures. By employing this more straightforward search space, the method can outperform the previous two approaches on various node classification tasks.

The previous methods employ reinforcement learning, while their focus is to design graph neural networks for node classification tasks. However, employing reinforcement learning approaches introduces complexity to the pipeline, as the agent's training must also be optimized. By utilizing an evolutionary algorithm developed in earlier chapters, we reduce the components that need fine-tuning. Furthermore, parallelization of evolutionary algorithms is straightforward [16], enabling our approach to be easily scaled up.

7.3 Methodology

This chapter employs the evolutionary algorithm described in chapter 6 to generate graph convolutional architectures. We evaluate the compiled networks on the relative ranking prediction task, utilizing architectures generated from both global as well as cell search spaces. We utilize the global search space on Fashion-MNIST (as in chapter 6) and the NASBench-101 benchmarking dataset. We first describe how the various genes are implemented, how the individuals are evolved, how a network is compiled, and how it is evaluated.

7.3.1 GCNs and the SAGE framework

Graph convolutional networks extend the basic concepts of conventional convolutional layers to datasets organized in a graph-like structure [17, 18]. For a graph $G = (E, V)$ with edges E and nodes V, we associate a feature vector H_i, of length N with each node. We can then aggregate feature tensors from the neighborhood of V_i utilizing permutation-invariant operations, such as element-wise max or average. This results in a new vector T_i of size N. Multiplying the vector by a weight matrix and applying a non-linear activation function to the result, a simple graph convolutional layer can be defined.

Although most early work on GCNs focused on transductive learning, GraphSAGE [19] focuses on inductive learning. It proposes the employment of trainable aggregators, aiming to achieve inductive learning capabilities. The two most exciting aggregators (also the ones used in our approach) are LSTMs [20] and Pooling networks. As LSTMs are not permutation-invariant, a random shuffling of the inputs ensures that the network will not over-fit on the data order. Pooling networks use a trainable weight matrix, which is applied individually to each neighboring node's feature vector, followed by a non-linear activation. Finally, the results are aggregated by an element-wise max pooling operation.

7.3.2 Genome representation

To represent our GCN architectures, we employ layer and connection genes, as in previous chapters, but also employ a hyper-parameter gene, describing specific network-wide attributes. The three chromosomes comprise a single genome, fully describing a GCN architecture.

Table 7.1: Available layer gene options.

Layer Selection	Aggregator	Filters
SAGE_LSTM.H	LSTM	Half Filters ($F/2$)
SAGE_POOL.H	POOLING	Half Filters ($F/2$)
SAGE_LSTM.N	LSTM	Standard Filters (F)
SAGE_POOL.N	POOLING	Standard Filters (F)
SAGE_LSTM.D	LSTM	Double Filters ($2F$)
SAGE_POOL.D	POOLING	Double Filters ($2F$)

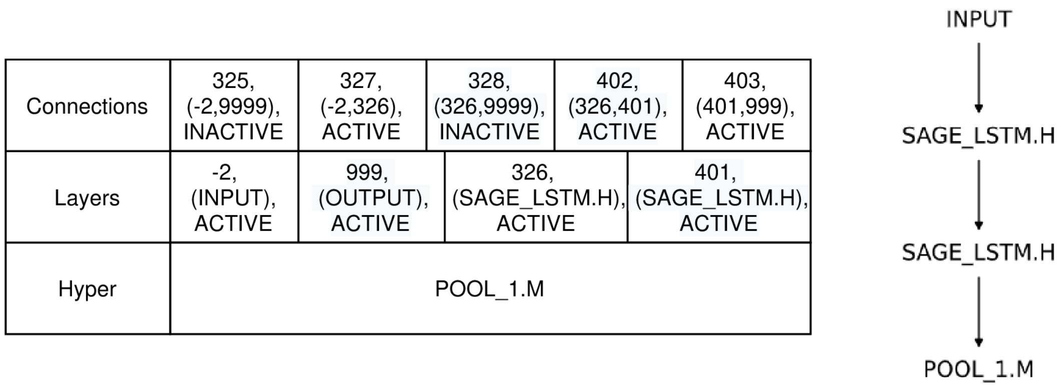
Layer and connection genes

As in the previous chapter, we employ a fixed number of channels for each network. However, instead of having only half-channel layer options, we also employ double-channel options. As such, employing GraphSAGE layers with LSTM or pooling aggregators, we have 6 distinct layer configurations; LSTM aggregators with half, double or normal channel number, and pooling aggregators with half double or normal channel number. Connection genes are implemented in the same way as in chapter 6.

Hyperparameter genes

Instead of having a simple output node in our networks, we employ a global pooling operation to create standardized-size embeddings, irrespective of each input graph's topology and size. As this layer significantly affects the architecture's performance, we implemented it as a hyper-parameter in the genome. The hyper-parameter gene indicates selecting this layer type and parameters from one of 5 individual options: a MaxPooling layer, an AveragePooling layer, and a SortPooling layer, with $k \epsilon$ [1, 2, 3] nodes retained.

Figure 7.1 depicts an example of the genome and the corresponding architecture. The first number in the gene indicates its first appearance (mutation number). The first number indicates the source for the connection chromosome while the second indicates the target node (expressed as the mutation number). The string in the parenthesis indicates the layer configuration selected. Finally,



Connections	325, (-2,9999), INACTIVE	327, (-2,326), ACTIVE	328, (326,9999), INACTIVE	402, (326,401), ACTIVE	403, (401,999), ACTIVE
Layers	-2, (INPUT), ACTIVE	999, (OUTPUT), ACTIVE	326, (SAGE_LSTM.H), ACTIVE	401, (SAGE_LSTM.H), ACTIVE	
Hyper	POOL_1.M				

Figure 7.1: **Left:** Example of a genome with connection, layer, and hyper-parameter chromosomes. **Right:** The corresponding network.

the single hyper-parameter gene indicates the global pooling selection for the architecture.

Mutation and Evolution

For the hyper-parameter genes, mutation induces a random permutation of the layer configuration. Connection genes are mutated in the same manner as before. Finally, layer gene mutation changes the layer configuration selection to another at random.

Evolution follows the same rules as in chapter 6. A population of P random genomes is initialized, with a single hidden GCN layer and a random global pooling layer. At each cycle, N candidates are randomly selected to reproduce, and the one with the highest fitness creates offspring by mutation. A gene is mutated with probability p_m and a connection gene is added with probability p_a. The oldest individual is discarded after the offspring is evaluated and inserted in the population.

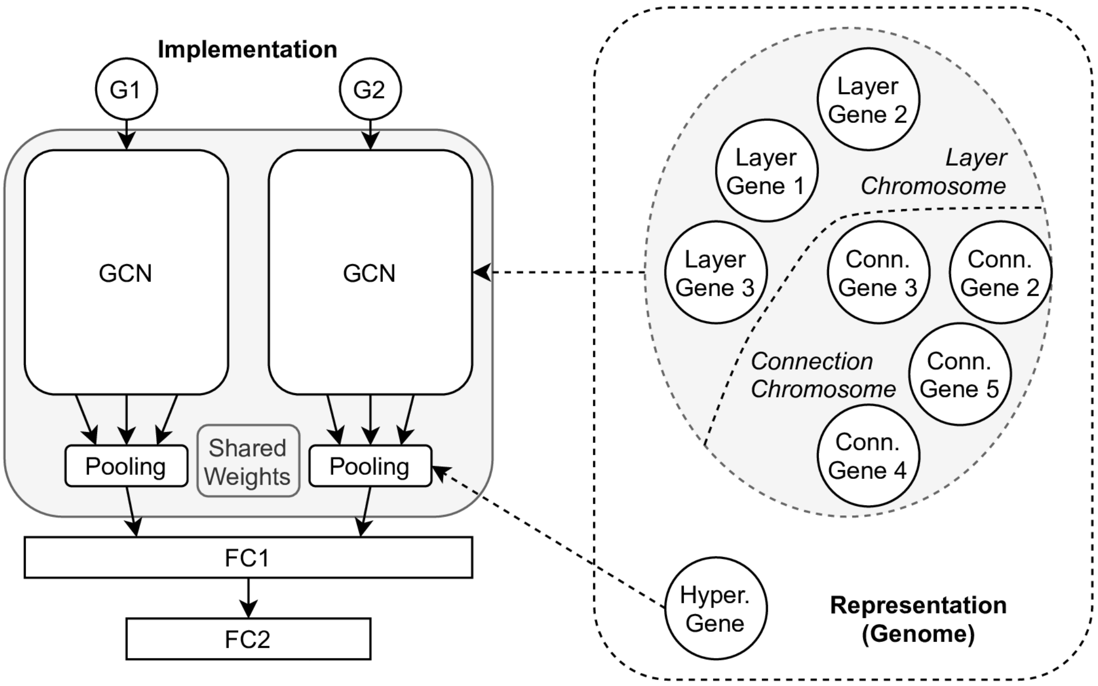


Figure 7.2: Genome to network implementation.

7.3.3 Graph ranking implementation

We employ a two-level dense head to implement graph ranking, which is fed by the GCN architecture. As our inputs are a pair of architectures G_1, G_2, the head receives double the features of a single GCN. We set the number of neurons equal to half of the GCN features for the first dense layer, while the second layer is the classification layer. We employ a ternary class, allowing the network to decide if the two architectures are equal($G_1 \approx G_2$), or one is better than the other ($G_1 > G_2$ or $G_1 < G_2$). We assume that two networks are equal if their performance is within a small margin e of one another. We choose to share weights between the two GCNs to reduce the number of trainable parameters and make the networks more robust. The two outputs are concatenated and then forwarded to the dense head. Figure 7.2 depicts the transition from a genome to an implemented network, summarizing this section's information.

7.3.4 Generating the datasets

For the NASBench-101 dataset, we sample at random a number of architectures. We generate their relative rankings by first splitting them into a train and test set and then testing for equality (if the absolute difference between their average accuracy is smaller than e). Note that there is no overlap between the train and test set. As such, when tested, the GCN network will try to predict the relative quality of previously completely unseen networks. The information regarding 108 epochs of training is utilized.

For the global search space dataset, we generate 980 random architectures for the Fashion-MNIST dataset, employing the same available layer configurations as in NASBench-101, namely 1x1 convolution layers with 16 filters 3x3 convolutions with 32 filters, and a 2x2 max-pooling layer. In addition, each network is trained for 108 epochs, using the AdamW optimizer[21]. Finally, the accuracy of the test set is recorded. Although we employ the same layer configurations as NASBench-101, we do not employ a fixed macro-architecture. As such, we have a global search space with limited options regarding layer setups. Nonetheless, we limit the available number of nodes to 8 to enable comparison with ReNAS, which requires a fixed maximum number of nodes as its inputs are in a fixed-length tensor format. The dense head in these networks consists of three dense layers; the first with a number of neurons equal to 1/4 of the incoming features, the second with half the neurons of the first, and finally, a classification layer with 10 neurons.

7.4 Experiments

7.4.1 Experimental setup

For the NASBench-101 architectures dataset, our experimental setup consists of populations of 100 individuals, evolved for a total of 400 cycles. At each cycle, 25 individuals are selected as parents, utilizing the generated network's Kendall's tau correlation coefficient [22] of the generated rankings as a fitness function.

Table 7.2: Experimental Parameters.

Parameter	Value	Parameter	Value
P	100	ϵ	0.01
N	25	train size	150
C	400	test size	350
p_m	0.25	optimizer	AdamW
p_a	0.25	fitness metric	Kendall's tau
train batch	256	train epochs	2
standard filters	300		

Each parent produces an offspring with 25% probability of adding a node to its architecture and a 25% probability of mutating a gene. We employ 300 channels as the standard number, thus having layers with 150, 300, and 600.

We train each network for 2 epochs on the AdamW optimizer, with a learning rate of $1.0e-3$ and weight decay of $1.0e-2$. During the search phase, we employ 500 architectures, split into a 70% train set and 30% validation set. The same setup was also employed in the Fashion-MNIST dataset. We set the equality margin $\epsilon = 0.01$. The DGL python library was incorporated into NORD to enable network compilation. All GCN experiments are executed on a single GTX1060 graphics card. The parameters are summarized in Table 7.2. To establish a baseline, we also implemented ReNAS, utilizing only the layer type and adjacency matrix, as this is also the only information available to the GCN models.

7.4.2 Experimental results

Following the 400 cycle evolution, the algorithm has successfully found a well-performing neural architecture relatively quickly. In the NASBench-101 dataset (as depicted in Figure 7.3), the initial population is able to incorporate individuals with ranking correlations above 0.825. Moreover, architectures that achieve over 0.91 correlation are also present during the last cycles. This level of ranking correlation is close to the re-sampling of training runs conducted in chapter 5.

On the fashion-MNIST dataset (Figure 7.4), it is evident that the networks

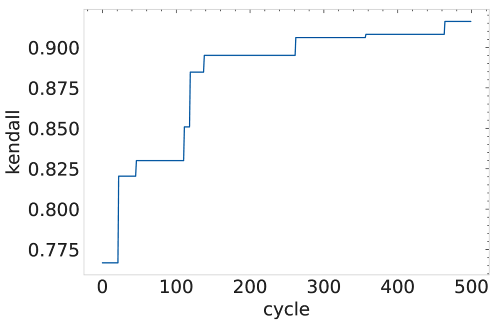


Figure 7.3: Progression of the best found architecture's performance on NAS-Bench-101.

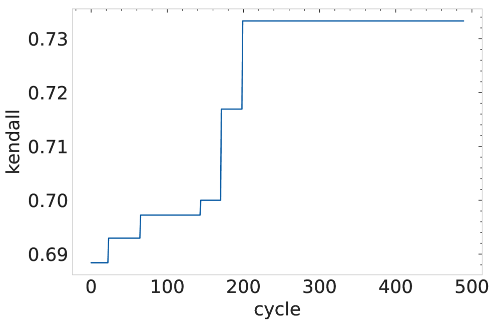


Figure 7.4: Progression of the best found architecture's performance on Fashion-MNIST

struggle to model the more complex search space. Nevertheless, the overall best Kendall's correlation coefficient is marginally over 0.73, which is still useful as it was seen in the Monte-Carlo experiments of chapter 5. Note that the search process required approximately 10 GPU hours on an NVIDIA GTX 1060 card (not accounting for the time required to generate the datasets).

Going further, it is interesting to explore the various GCN architectures generated. The boxplots in Figures 7.6, 7.5 indicate that MaxPooling layers provide better performance as global aggregators, while the SortPooling layers benefit more from higher numbers of retained features ($k = 3$). This observation is even more apparent in the global search space dataset. Nonetheless, MaxPooling seems also to induce a high variation in network performance. This variation can partly be attributed to the sheer number of architectures employing MaxPooling.

As we employ a very small number of training epochs, we expect that highly parameterized models will struggle to optimize their weights, and models with fewer parameters will survive in the population longer. Admittedly, in Figures 7.7, 7.8, two scatterplots with Kendall's correlation coefficient on the y-axis and the number of parameters on the x-axis confirms our hypothesis. Moreover, by color-grading the points to reflect correlation coefficients for the top-50% of the architectures, we notice that most of the architectures have less than 200,000 parameters. For NASBench-101, we observe that models with slightly more parameters perform better in the top-50% sub-set. In Fashion-MNIST, a higher

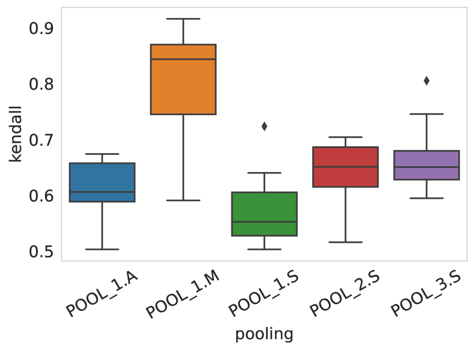


Figure 7.5: Performance boxplots for NAS-Bench-101, grouped by final aggregation layer.

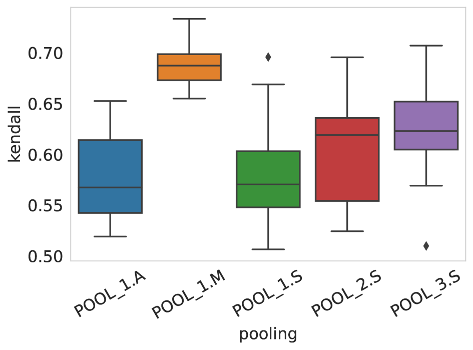


Figure 7.6: Performance boxplots for Fashion-MNIST, grouped by final aggregation layer.

noise level exists, as the dataset is considerably harder to model.

Best architectures and fine-tuning

To analyze further the optimal architectures discovered in NASBench-101, we further trained the architectures evaluated at least five percent of the overall architectures number (25 times). We select the architecture with the highest combined mean accuracy and Kendall's correlation coefficient from this pool of frequently evaluated architectures. The chosen architecture consists of two sequentially connected SAGE_LSTM.H layers, followed by a MaxPooling layer. The network has 330,000 trainable parameters. We try various optimization schemes, although our primary goal is to evaluate the architecture's generalization ability. We utilize both 70%/30% train/test splits and 30%/70%, NASBench samples of 500 and 1500 architectures, and an equivalence threshold of $e = 0.003$, thus increasing the difficulty, as a pair of architectures must perform very closely to classify as equivalent. The performance of the network under various training setups for the NASBench-101 dataset is depicted in Table 7.3. Setting the batch size to 32 and allowing for a more granular training, the network is able to achieve a correlation coefficient of over 0.95. The network retains this performance even when we require a difference in accuracy of less than 0.3% to consider two architectures equivalent. Employing only 30% of the dataset as a training set is

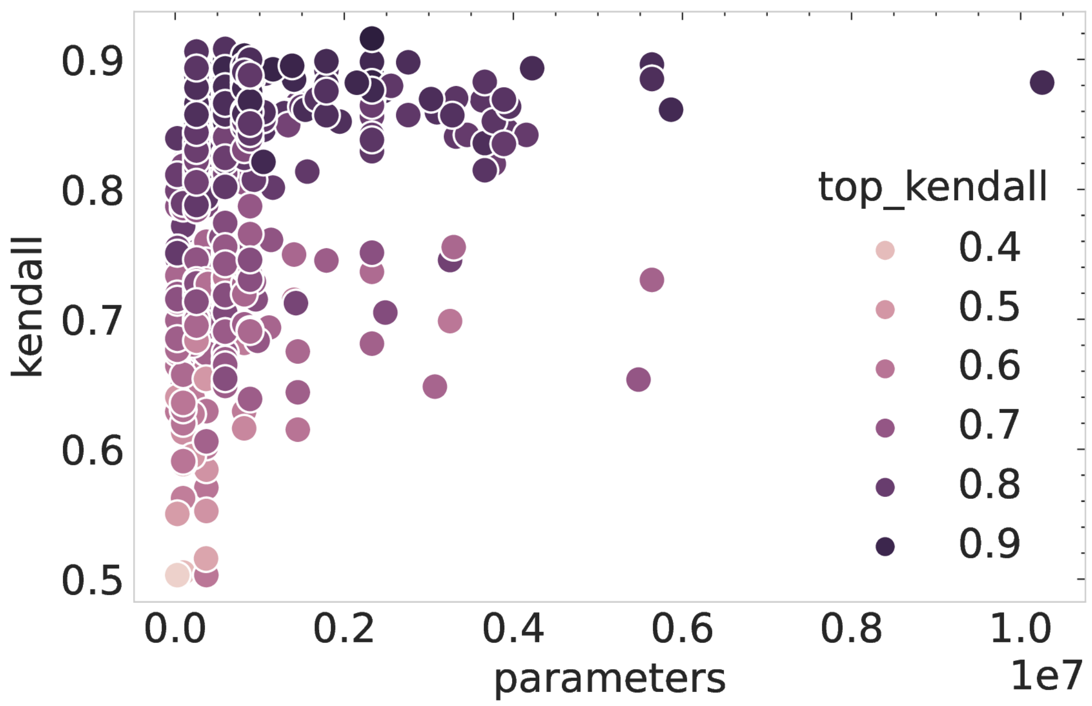


Figure 7.7: Scatterplot of kendall's tau (y-axis) vs number of parameters(x-axis) on NAS-Bench-101. Color indicates kendall's tau for the top-50% test set instances.

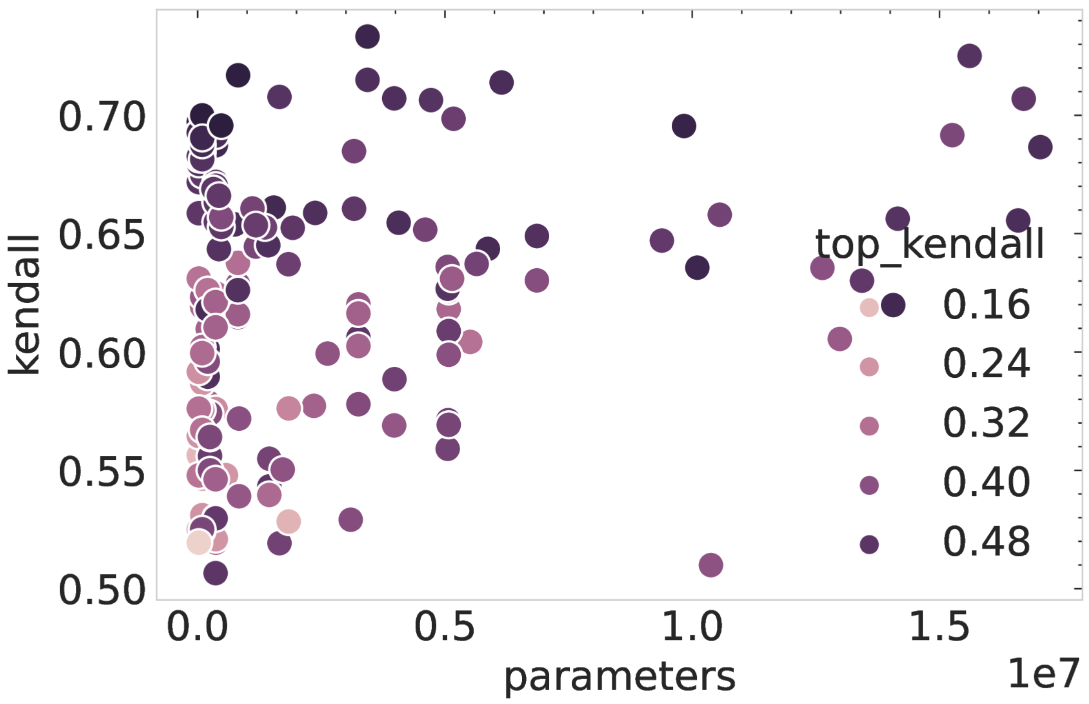


Figure 7.8: Scatterplot of kendall's tau (y-axis) vs number of parameters(x-axis) on Fashion-MNIST. Color indicates kendall's tau for the top-50% test set instances.

Table 7.3: Fine-tuning results on NAS-Bench-101.

Train Set	Test Set	ϵ	Kendall τ	Top-50% τ
350	150	0.01	0.954	0.95
1050	450	0.01	0.93	0.904
350	150	0.003	0.898	0.886
1050	450	0.003	0.932	0.934
450	1050	0.003	0.907	0.853

sufficient to train the network, achieving a 0.907 correlation coefficient while also retaining a high top-50% correlation, thus exhibiting satisfactory generalization ability.

For the architecture derived from the Fashion-MNIST dataset, the chosen network has a significantly more complex architecture, as depicted in Figure 7.9. The architecture is able to achieve a correlation coefficient of $\tau_{total} = 0.725$ for the whole test set and $\tau_{0.5} = 0.489$ for the top-50% of the architectures. We also employ the last training setup used in the NASBench-101 model of 30%/70% train/test split. Nonetheless, the network cannot learn meaningful representations, resulting in a degradation in performance. Employing a One-Cycle learning rate policy [23] with a maximum learning rate of 1.2e-3 the network is able to achieve $\tau_{total} = 0.817$ for the whole test set and $\tau_{0.5} = 0.749$ for the top-50%.

7.4.3 Comparing to ReNAS

As we aim to evaluate our NAS method, we implement ReNAS, which has achieved significant out-of-sample performance on the NASBench-101 architectures while outperforming other methods. As our approach only utilizes architecture structure, we implement the simplest version of ReNAS, where cell connectivity and layer types are utilized as features. This approach ensures a fair comparison between ReNAS and our best network (EGraph). To train the ReNAS model, we use Adam, with a learning rate of $1e-3$ and weight decay of $5.0e-4$ for 200 epochs, according to [5]. Margin Ranking Loss is utilized, which yielded promising results in the original paper when the simple feature space was

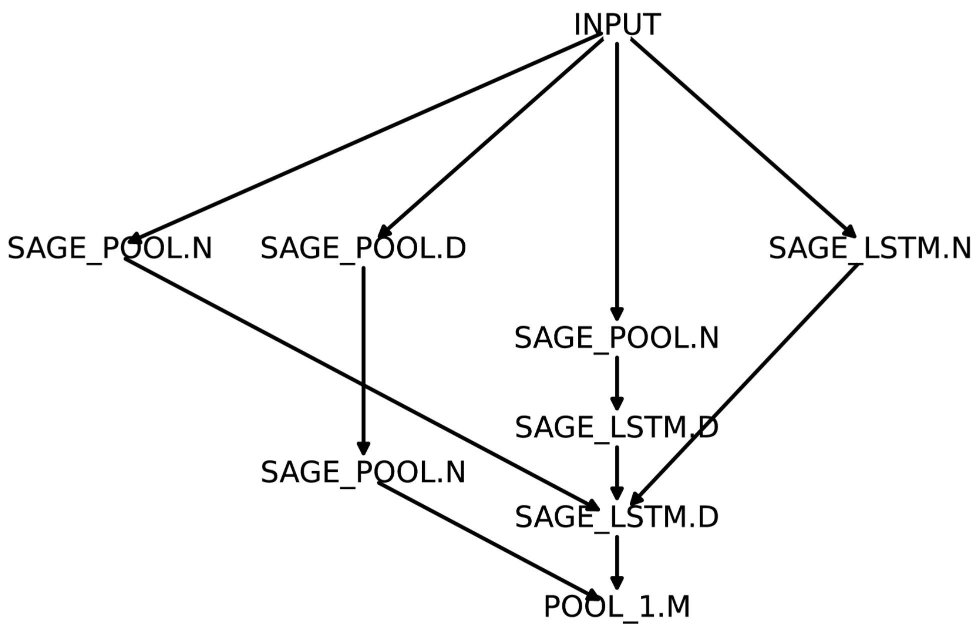


Figure 7.9: Best architecture found for the Fashion-MNIST derived dataset.

Table 7.4: Comparison with ReNAS on NAS-Bench-101.

Model	Batch Size	Train/Test	ϵ	τ	Top-50% τ
ReNAS	32	450/1050	0.003	0.828	0.57
ReNAS	1024	450/1050	0.003	0.856	0.64
EGraph-Regression	32	450/1050	0.003	0.872	0.71
EGraph	32	450/1050	0.003	0.907	0.853
EGraph 5 Epochs	32	450/1050	0.003	**0.922**	**0.895**

used. We attempted to train ReNAS with batch sizes of both 1024, as proposed by the authors, as well as 32, which is what our model uses. Furthermore, we modify our dense head's final layer to a single neuron to test EGraph's regression performance, using Margin Ranking Loss to optimize it and 20 training epochs. Finally, we attempt to employ bigger training budgets for EGraph, increasing the training epochs to 5.

Table 7.4 summarizes the resulting performances for NAS-Bench-101. For the Fashion-MNIST dataset, the best performance obtained by ReNAS (batch size of 1024, 200 training epochs) achieved $\tau_{total} = 0.628$ and $\tau_{0.5} = 0.333$. EGraph's best-performing version (without One Cycle Learning Rate Policy, i.e. batch size of 32, 5 training epochs) achieved $\tau_{total} = 0.802$ and $\tau_{0.5} = 0.720$.

Experimental results show that ReNAS has solid performance, especially considering that the network's architecture consists of two 2-Dimensional convolutions, followed by a batch normalization layer. However, EGraph with the regression head can marginally outperform ReNAS in both the top-50% of the architectures and the whole dataset. This behavior can be attributed to EGraph's use of convolutions designed explicitly for the graph domain, while ReNAS employs conventional convolutional layers coupled with innovative feature engineering.

EGraph with the binary ranking head can outperform all other models, retaining high correlation in both the whole dataset as well as the top-50% of the architectures. Furthermore, it is interesting that ReNAS performs better when a large batch size is employed, while EGraph requires small batch sizes. Finally, for the Fashion-MNIST dataset, although EGraph manages to outperform ReNAS, employing a One Cycle Learning Rate Policy significantly boosts performance, further confirming the specific learning rate schedule merit.

7.5 Limitations

Even though our approach outperformed a solid baseline (ReNAS), deriving the architectures is significantly more expensive. Furthermore, training the architecture is also more expensive due to differences in the datasets. For example, for a regression task with N graphs, N instances are generated. For binary ranking tasks, the resulting dataset is of size N^2. In cases where we do not need a symmetrical dataset, as is EGraph, the total number of comparisons is $(N * N + 1)/2$. This increase in training instances dramatically increases the computational resources required to train a model.

Our search space is limited to a relatively small number of layers and pooling operations selection concerning the search method. From the experimental results, a de-facto MaxPooling global pooling layer would be beneficial, allowing us to further expand the layer configurations without increasing the absolute search space size while also improving the diversity of the architectures.

7.6 Conclusions

In this chapter, we propose a method for searching for graph convolutional architectures for the task of modeling candidate architecture performance. Expanding on previous chapters, we examine a method for generating GCNs as performance predictors. Compared to other works that inspired us to work in this direction, we are able to produce better predictive models. We observe that for our proposed search space, MaxPooling operations outperform other global pooling methods;it could be utilized as the de-facto aggregator in similar works, thus significantly reducing the search space size.

Due to the high computational burden of evaluating complex GCN architectures, we restrict our search space to include a limited number of layer configurations, while we employ a reduced epoch candidate evaluation method, employing only 2 training epochs. Nonetheless, by applying our method to architectures derived from the NASBench-101 and Fashion-MNIST datasets, we are able to outperform ReNAS when the input features contain similar information. Although able to outperform ReNAS, our approach also has the advantage of not requiring specific feature engineering to be able to process neural architectures.

Chapter 7 References

[1] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, 7 2019.

[2] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312.

[3] D. Long, S. Zhang, and Y. Zhang, "Performance prediction based on neural architecture features," IEEE, pp. 77–80, 2019.

[4] Y. Tang, Y. Wang, Y. Xu, H. Chen, B. Shi, C. Xu, C. Xu, Q. Tian, and C. Xu, "A semi-supervised assessor of neural architectures," pp. 1810–1819, 2020.

[5] Y. Xu, Y. Wang, K. Han, S. Jui, C. Xu, Q. Tian, and C. Xu, "Renas: Relativistic evaluation of neural architecture search," *arXiv*, pp. arXiv–1910, 2019.

[6] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural architecture search with bayesian optimisation and optimal transport," pp. 2016–2025, 2018.

[7] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," pp. 19–34, 2018. [Online]. Available: http://github.com/tensorflow/

[8] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," *Advances in Neural Information Processing*

Systems, vol. 2018-December, pp. 7816–7827, 8 2018. [Online]. Available: http://arxiv.org/abs/1808.07233

[9] T. Chau, Ł. Dudziak, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane, "Brp-nas: Prediction-based nas using gcns," *arXiv preprint arXiv:2007.08668*, 2020.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[11] B. Deng, J. Yan, and D. Lin, "Peephole: Predicting network performance before training," *arXiv preprint arXiv:1712.03351*, 2017.

[12] Y. Sun, H. Wang, B. Xue, Y. Jin, G. G. Yen, and M. Zhang, "Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 350–364, 2019.

[13] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu, "Graphnas: Graph neural architecture search with reinforcement learning," *arXiv preprint arXiv:1904.09981*, 2019.

[14] K. Zhou, Q. Song, X. Huang, and X. Hu, "Auto-gnn: Neural architecture search of graph neural networks," *arXiv preprint arXiv:1909.03184*, 2019.

[15] H. Zhao, L. Wei, and Q. Yao, "Simplifying architecture search for graph neural network," *arXiv preprint arXiv:2008.11652*, 2020.

[16] J. Kacprzyk and W. Pedrycz, *Springer handbook of computational intelligence.* Springer, 2015.

[17] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," Springer, pp. 593–607, 2018.

[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[19] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” pp. 1024–1034, 2017.

[20] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *arXiv preprint arXiv:1711.05101*, 2017.

[22] M. G. Kendall, “A new measure of rank correlation,” *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[23] L. N. Smith, “Cyclical learning rates for training neural networks,” in *2017 IEEE winter conference on applications of computer vision (WACV)*. IEEE, 2017, pp. 464–472.

Chapter 8

Parallelizing Differentiable Neural Architecture Search

Previous chapters studied the acceleration of various distributed genetic and evolutionary approaches to NAS. Although these methods have their merit and are indispensable when searching global search space, differentiable methods have produced state-of-the-art models for various datasets [1, 2, 3, 4]. There have been many extensions of the original paper, DARTS, but as with most other NAS research, they focus on improving DARTS' computational and memory demands and its effectiveness in generating competent architectures. Being a NAS method, DARTS is computationally expensive and can benefit from parallel and distributed computing. In this chapter, we implement and study a distributed version of DARTS, which is the first such implementation to the best of our knowledge. Due to the pruning of connections before producing the final architecture, the noise induced by parallelizing the method should not significantly affect the results. We aim to investigate this hypothesis by implementing and running both serial as well as parallel versions of DARTS

8.1 Introduction

One of the most popular and successful differentiable NAS methodologies is DARTS[5] (presented more in-depth in chapter 2). Extensions of this method have led to the discovery of the current state-of-the-art network for the Fashion-MNIST data set [4]. However, when compared to other NAS methods, one of the main disadvantages is the significantly larger memory requirement due to the need for the whole supernet to be in the CPU memory at all times and interim results needed for the backpropagation of weight and architecture errors. This requirement indirectly poses an upper limit on the maximum size that DARTS can utilize. Furthermore, the parallelization of differentiable methods is usually not straightforward.

Given that DARTS is a differential method, asynchronous parallel implementation of gradient descent could be applied in theory [6, 7, 8]. The problem lies in the special handling of the optimization process and the noise introduced to the training process. The most straightforward, synchronous approach would be to have all parallel copies of the network average their weights after each epoch [9]. Although there is a possibility of converting to local optimal for non-convex models, neural networks show an increasingly convex loss surface landscape as the number of their parameters increases [10].

This chapter aims to explore the behavior of first-order parallel DARTS under synchronous parallel SGD settings. We employ 1,2 and 4 parallel workers on the Fashion-MNIST dataset. The noise expected to induce parallel training should only affect the search phase. Furthermore, as the final architecture is pruned, minor variations in the architecture weights should not dramatically alter the final architecture. We first explain our methodology and our experimental results. Finally, we present the limitations and findings of our research.

8.2 Methodology and Experimental Setup

As we aim to explore the behavior of DARTS under data-parallel training, we employ a synchronous approach, hoping to curtail its impact on a bi-level optimization problem. To minimize communication overhead (and thus increase speed), we perform a single all-reduction operation (averaging) at the end of each training epoch (Algorithm 1). All workers must thus wait for anyone lagging to proceed to the next epoch. When compared to a more conventional synchronous approach, for example, gradient aggregation (Algorithm 2), we reduce the communication overhead significantly. Specifically, the communications are reduced by N/B, where N is the number of training examples, and B is the batch size. In our application dataset, Fashion-MNIST, with a batch size of 64, the reduction factor is a significant 937,5; as for DARTS, the whole training dataset is utilized for training (both training as well as validation sets). Moreover, by reducing communication, we also reduce the possible cases where strugglers may occur [9].

Algorithm 1: Weight-based data-parallel DARTS, worker k

```
Data: $L_{train}, L_{valid}, n_{batches}, n_{epochs}$
Result: $a$
$w \leftarrow$ broadcast($w$) from worker 0;
$a \leftarrow$ broadcast($a$) from worker 0;
for $e = 0, 1, ..., n_{epochs}$ do
    for $i = 0, 1, ..., n_{batches}$ do
        $train \leftarrow L_{train}[i]$; $valid \leftarrow L_{valid}[i]$;
        $G_w \leftarrow \frac{\partial Error(train; w,a)}{\partial w}$;
        $w \leftarrow w - lr * (G_w)$;
        $G_a \leftarrow \frac{\partial Error(valid; w,a)}{\partial a}$;
        $a \leftarrow a - lr * (G_a)$;
    end
    $w \leftarrow$ allreduce($w$);
    $a \leftarrow$ allreduce($a$);
    /* Wait for allreduce to complete                          */
end
```

The synchronous DARTS approach is implemented in NORD, utilizing NVIDIA's NCCL backend for collective communications. NCCL allows data transmission directly from GPU memory, thus alleviating the need to transfer data to CPU memory. We employ a proxy search space during the search phase, consisting of 16 initial channels and 8 cells. Layer weights are optimized with Stochastic Gradient Descent, employing an annealing learning rate schedule with an initial learning rate of 0.025 and a minimum learning rate of 0.001. Architecture weights are optimized utilizing the Adam optimizer. For each MixOp layer, we employ the original 8 operations; zeroize, identity, max and average pooling with kernels of size $3x3$, separable convolutions with kernel sizes $3x3$ and $5x5$, and dilated convolutions with kernel sizes $3x3$ and $5x$. For dataset augmentation, we employ the standard NORD Fashion-MNIST implementation, consisting of random horizontal flips and random erasing. The experiments are conducted in cloud clusters of 1, 2, and 4 NVIDIA TESLA A100 GPUs. Each experimental setup is executed for 4 distinct runs with a different initial seed to accommodate the fact that initial architecture and layer weights can both favor and disfavor a particular run.

Algorithm 2: Gradient-based data-parallel DARTS, worker k

```
Data: $L_{train}, L_{valid}, n_{batches}, n_{epochs}$
Result: $a$
$w \leftarrow$ broadcast($w$) from worker 0;
$a \leftarrow$ broadcast($a$) from worker 0;
for $e = 0, 1, ..., n_{epochs}$ do
    for $i = 0, 1, ..., n_{batches}$ do
        $train \leftarrow L_{train}[i]$; $valid \leftarrow L_{valid}[i]$;
        $G_w \leftarrow \frac{\partial Error(train;w,a)}{\partial w}$;
        $w \leftarrow w - lr*$allreduce($G_w$);
        /* Wait for allreduce to complete */
        $G_a \leftarrow \frac{\partial Error(valid;w^*,a)}{\partial a}$;
        $a \leftarrow a - lr*$allreduce($G_a$);
        /* Wait for allreduce to complete */
    end
end
```

After the search phase, we retain the best architecture of each GPU setup (1, 2, and 4). Then, we train the final extracted architectures for 100 epochs to compare their performances. In this final phase, we employ an expanded outer skeleton, consisting of 36 channels and 20 cells, aiming to increase the number of parameters in the networks (in line with the original work of [5]). The validation set is only 10% of the training set during this training phase, while Stochastic Gradient Descent with momentum optimizes the network weights. Table 8.1 depicts the parameters for the search and final training phases.

Table 8.1: Search and training parameters

Search Phase		Training Phase	
Layer Optimizer	SGD	Layer Optimizer	SGD
Learning Rate	0.025-0.001	Learning Rate	0.025
Momentum	0.9	Momentum	0.9
Architecture Optimizer	Adam	-	
Channels	16	Channels	36
Cells	8	Cells	20
Layer Types	8	-	
Batch Size	64	Batch Size	96
Validation Percentage	50%	Validation Percentage	10%
Epochs	50	Epochs	100

8.3 Results

In this section, the results for all approaches are presented. First, we analyze the behavior of the serial and the parallel implementations. Following, we compile the extended networks, and by fully training them, we extract information regarding the viability of employing synchronous data-parallel approaches for DARTS.

8.3.1 Search Phase

As the results from the training dataset indicate, serial implementations of DARTS outperform multi-GPU parallel implementations. Both the training and validation accuracy is higher for the serial version. Furthermore, final train and validation accuracy are also higher for single-GPU runs. The differences are even more prominent in training accuracy (Figure 8.1), rather than validation accuracy (Figure 8.2). In validation accuracy terms, all runs could produce results within a 1% accuracy range, namely in [92.2%, 93.2%].

Nonetheless, these results do not concern the final network and only indicate search progress. As such, they cannot be directly compared. Speedup seems to scale almost linearly with the number of GPU workers. The two-GPU setup achieves a speedup of 1.82, while the four-GPU setup (Table 8.2 achieves a

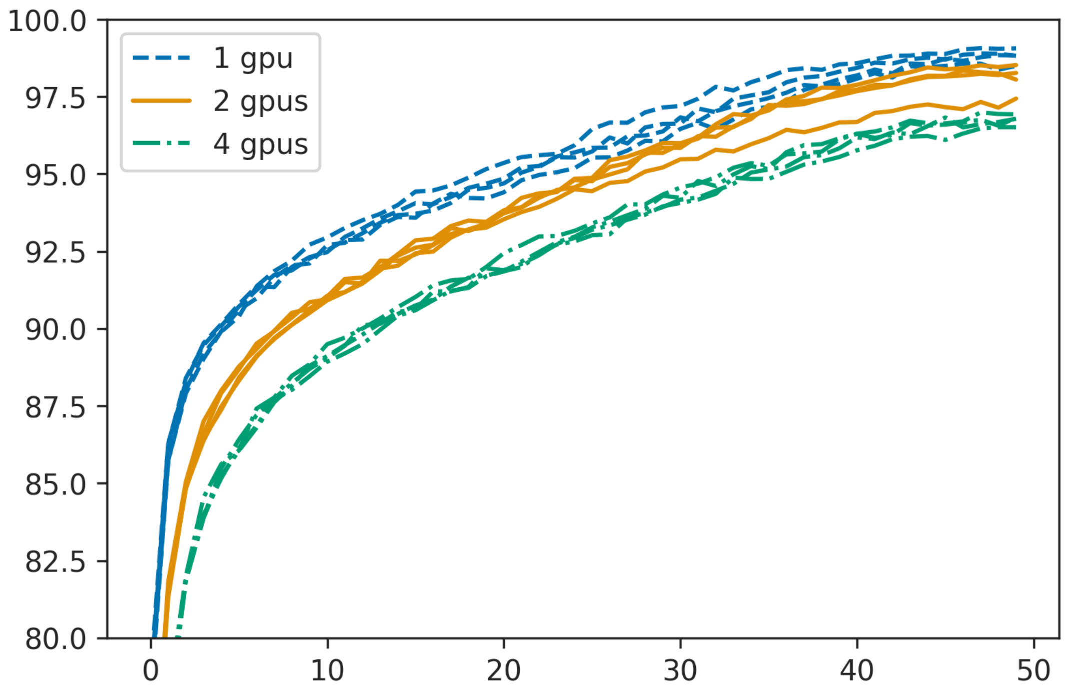


Figure 8.1: Training accuracy curves for the search phase.

speedup of 3.2. The efficiency of the implementation is thus 90% for the two-GPU and 80% for the four-GPU setup. This can probably be attributed to straggles, as communication overhead is minimal.

Backup workers [9] could, in theory, alleviate the problem of stragglers, but this would require an additional GPU performing work. If we add another worker in the four-GPU setup and achieve a speedup of 4, efficiency would remain at 80%. Assuming that a 10% efficiency is lost for every doubling of the GPU workers, at 8 nodes, the efficiency would drop at least to 70%. With 8+1(backup) workers and a speedup of 8, the efficiency would be at 88%, justifying its employment.

Table 8.2: Key performance metrics

GPUs	Best Train Accuracy	Best Val. Accuracy	Avg. Time	Speedup
1	99.06%	93.19%	26708s	-
2	98.52%	92.99%	14659s	1.82
4	96.92%	92.63%	8393s	3.18

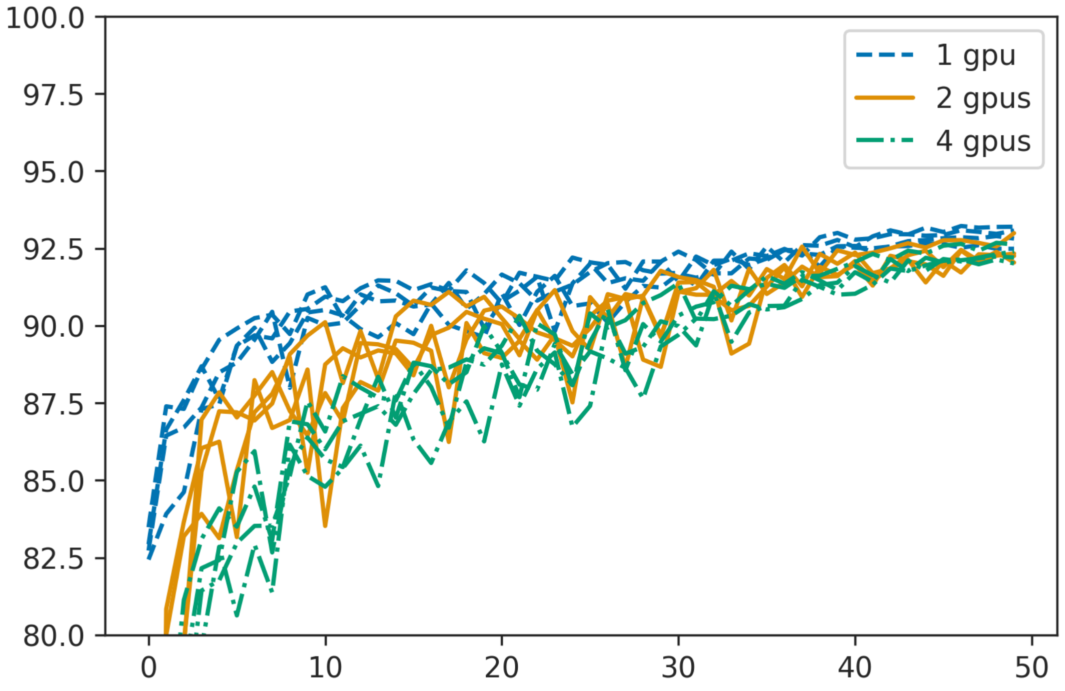


Figure 8.2: Validation accuracy curves for the search phase.

To examine differences between the architectures extracted from each run, we calculate the graph edit distance (GED) for intra-group and inter-group results between normal (Figure 8.3) and reduction cell architectures (Figure 8.4). Graph edit distance is calculated utilizing both topologies as well as layer selection. The same computational graph structure with a single difference in operation selection would give a graph edit distance of 2. First, the wrong operation should be removed, and then the correct operation should be added, resulting in two "moves". We observe that intra-group GED values (1, 2, and 4 GPUs) exhibit smaller deviation than inter-group GED values (1 vs. 4, 1 vs. 2, and 2 vs. 4 GPUs).

Applying a Kruskal-Wallis H test [11] for the normal cells yields a p-value of 0.4 while applying it to the reduction cell GEDs yields a p-value of 0.29. Consequently, we cannot say that any group produced significantly different distributions of architectures. Given that the mean GED value is 8 and each node has exactly one connection with all previous nodes, we can say that, on average, layer selections differ in four points, while the cells are computationally similar (Figure 8.2). As such, the synchronous data-parallel approach produces, on average, the same deviation in generated architectures, as the serial implementation of DARTS. From Figure 8.4 we can even see a slight tendency for the serial implementation to produce higher intra-group GED.

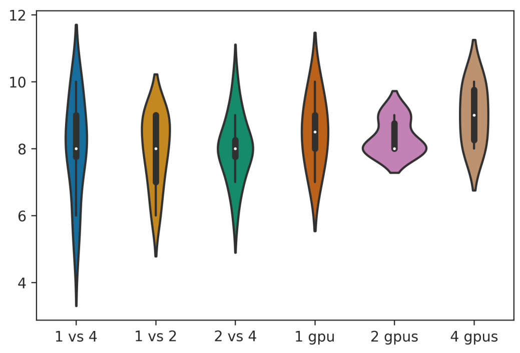


Figure 8.3: Graph edit distance values for the normal cells.

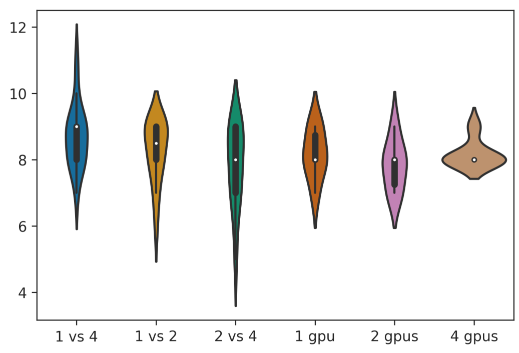
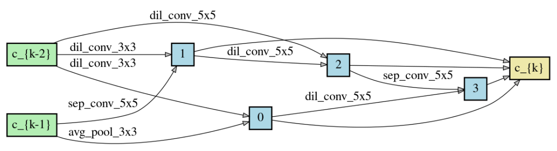


Figure 8.4: Graph edit distance values for the reduction cells.

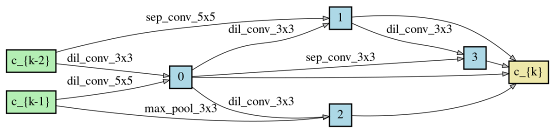
8.3.2 Best Model Analysis

Training the final architecture generated by each group entails increasing the compiled network's parameters number and optimizing it for 100 epochs. The architectures for normal and reduction cells in these networks are depicted in Figure 8.5. We observe that the inputs feed mainly convolutions for normal cells, with a minimal amount of pooling operations present. It is interesting to note that although all groups produced final architectures with mainly pooling operations in their cells, they were not the best-performing ones on our specific dataset.

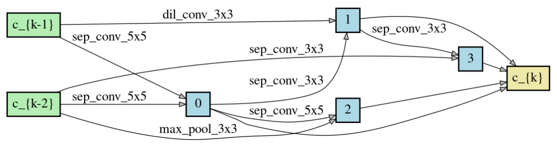
Even though the serial implementation exhibited the best in-search performance, the final architecture proposed by the serial groups underperforms in training set accuracy. Contrary, the other two networks follow a more similar trajectory (Figure 8.6). Nonetheless, all three networks have comparable behavior on the validation set, although a more thorough examination of their similarity should be conducted.



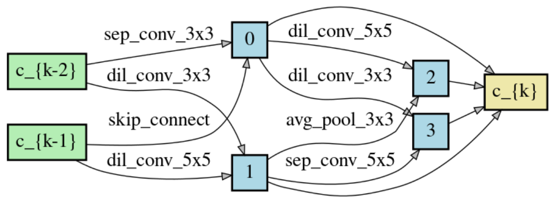
(a) 1 GPU normal



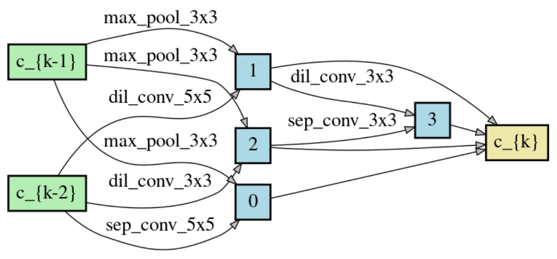
(b) 1 GPU reduction



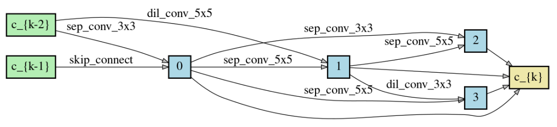
(c) 2 GPU normal



(d) 2 GPU reduction



(e) 4 GPU normal



(f) 4 GPU reduction

Figure 8.5: Best models cell architectures.

Assuming that the results of the first 10 training epochs rely heavily on the quality of the initial weight values, we apply a Kolmogorov-Smirnov test [12] for equality of distributions on the validation accuracies from epoch 10 to 100. Following, we correct for false discovery rate, utilizing the Benjamini/Hochberg correction [13]. Finally, table 8.3 depicts the results, which do not allow the rejection of the null hypothesis (the distributions are the same). As such, this further strengthens our observation from Figure 8.7, which shows a remarkably similar trend (with noise) for all networks.

Table 8.3: Kolmogorov-Smirnov and Benjamini/Hochberg-corrected p-values

Model	Kolmogorov-Smirnov	Benjamini/Hochberg-corrected
1 vs 2-GPUs	0.51	0.71
1 vs 4-GPUs	0.08	0.15
2 vs 4-GPUs	0.87	0.88

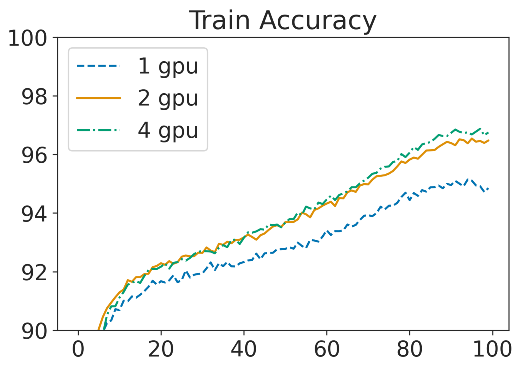


Figure 8.6: Best model training fine-tune accuracy.

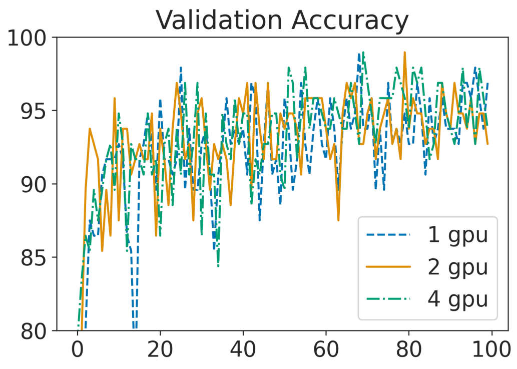


Figure 8.7: Best model validation fine-tune accuracy.

8.4 Limitations

Although we attempt to study the behavior of synchronous data-parallel DARTS to the best of our abilities, this chapter's main limitation is the small number of runs for each setup, as well as the lack of more GPU nodes to study the behavior on larger scales. Nevertheless, the results seem stable, although an increased number of executions for each setup would provide more insights into the properties of a data-parallel DARTS approach.

8.5 Discussion and Future Work

This final chapter of the thesis studies the behavior of Differentiable Architecture Search [5] under a data-parallel synchronous training approach. Employing 1, 2, and 4 GPU workers, we observe a speedup close to linear, although a 10% drop in accuracy is observed every time the number of workers doubles. We thus conclude that a back-up worker [9] would be beneficial when more than four GPU workers are employed. Moreover, it seems that parallelizing the method retains the stability levels of the original method, as the diversity between generated architectures remains unaffected. Analyzing graph edit distances of generated architectures further validates this observation.

The final networks (expanded to larger internal dimensions and depths) also exhibit similar performance. When trained, the trajectory of their validation accuracy follows the same trajectory, and the accuracy distributions do not differ at the 0.05 and 0.1 significance levels. This indicates that these networks are computationally equivalent. Any differences observed during the initial training epochs can be attributed to initial conditions, states of random number generators, and GPU instructions execution's inherent stochasticity.

Concluding, in this final chapter of this thesis, we show that a synchronous data-parallel approach is viable for DARTS. Furthermore, it results in similarly performing architectures, while the distributions of generated cell structures do not differ significantly.

Chapter 8 References

[1] N. Nayman, A. Noy, T. Ridnik, I. Friedman, R. Jin, and L. Zelnik, "Xnas: Neural architecture search with expert advice," in *Advances in Neural Information Processing Systems*, 2019, pp. 1975–1985.

[2] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong, "Pc-darts: Partial channel connections for memory-efficient architecture search," *arXiv preprint arXiv:1907.05737*, 2019.

[3] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[4] M. S. Tanveer, M. U. K. Khan, and C.-M. Kyung, "Fine-tuning darts for image classification," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 4789–4796.

[5] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *7th International Conference on Learning Representations, ICLR 2019*, 6 2018. [Online]. Available: http://arxiv.org/abs/1806.09055

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[7] S. Maleki, M. Musuvathi, and T. Mytkowicz, "Parallel stochastic gradient descent with sound combiners," *arXiv preprint arXiv:1705.08030*, 2017.

[8] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, and C. Ré, "Cyclades: Conflict-free asynchronous machine learning," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[9] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[10] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," in *Artificial intelligence and statistics.* PMLR, 2015, pp. 192–204.

[11] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[12] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[13] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal statistical society: series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

Chapter 9

Conclusions and Future Directions

This thesis aims to implement a Neural Architecture Search Framework and study parallel and distributed methods for NAS. Below the conclusions extracted during its course are summarised and possible future directions.

9.1 Conclusions

Chapter 3 proposed NORD, a NAS framework and library implemented in python. Having a standardized environment with easy to setup distributed training helped immensely with the experiments presented in the rest of the thesis. Furthermore, it enabled straightforward comparison between methods and variations of such methods. Given the speed at which NAS and deep learning advance, it required several significant refactors and feature integration, which proved more challenging than initially estimated.

Chapter 4 investigated the widespread (by NAS researchers) utilization of reduced training epochs as a proxy task for candidate architecture evaluation. We examined architectures generated for the CIFAR-10 dataset under reduced training epoch schemes. We received positive feedback by calculating rank correlation coefficients and applying a Monte-Carlo simulation, motivating us to investigate the approach further. Nonetheless, the architectures were generated on machines with limited memory and, as such, were relatively small.

Chapter 5 built upon the experience gained in chapter 4 and further expanded the investigation. More diverse architectures and more powerful hardware were employed to study relative ranks' behavior when reduced training was employed. Furthermore, architectures from the NASBench-101 dataset were investigated, and the rank correlation levels observed in both search spaces were translated to a noisy Rastrigin function search space. Results indicated that it is preferable to allocate more resources to the search algorithm rather than the candidate evaluation method. This is since we never directly measure the quality of a neural architecture. Instead, we observe the quality of a compiled network derived from the architecture.

Chapter 6 employs a distributed, regularized evolution approach along with the knowledge regarding reduced training evaluation to find optimal architectures in a Fashion-MNIST global search space. As a result, generated architectures can outperform other global search space NAS methods when evaluated similarly. Furthermore, it is observed that severely restricting the training epochs results in final networks that converge faster than comparable architectures. This can be attributed to larger networks requiring more epochs to distinguish themselves from the rest.

Chapter 7 applies the distributed method from chapter 6 to graph convolutional networks. Aiming to generate a predictive model for relative neural network quality, we generate GCN architectures and evaluate them on architecture datasets derived from the NASBench-101 and Fashion-MNIST datasets. Compared to the state-of-the-art method for predicting network quality, ReNAS, our generated models achieve better prediction quality. Furthermore, for our GCN search space, we observe that MaxPooling is the best choice as a global pooling layer and could possibly be employed as the de-facto choice/

Chapter 8 investigates a synchronous data-parallel approach for training a DARTS hypernet. By avoiding more traditional distributed SGD approaches, we reduce the communication overhead significantly and observe an almost linear speedup. We conclude that backup workers should be used for setups with more than four GPU workers. The parallel DARTS implementation itself produces architectures whose distributions match those of the original method. Moreover, when compiled into expanded networks, the final architectures have similar behavior to the architectures generated by the serial implementation. Distributing the batch to many workers also enables the evaluation of larger architectures, as the memory constraints are expanded.

9.2 Future Directions

Following the work in this thesis, there are some exciting outlooks. First, NAS seems to be able to produce significantly well performing architectures without human intervention. Second, recent work concerning transformer architectures have enabled many novel tasks to be automated. As such, employing parallel and distributed methods for transformer-specific NAS is a possible direction. Another possible direction would be the employment of NAS for reinforcement learning problems. If we view RL agents as individuals we could very well evolve a population of agents, each with a different "brain" or neural architecture. Finally, on a more technical aspect, Tensor Processor Units could be examined as a viable platform for parallel NAS.