The Dissertation Committee for Jason Zhi Liang
certifies that this is the approved version of the following dissertation:

Evolutionary Neural Architecture Search for Deep Learning

Committee:

Risto Miikkulainen, Supervisor

Ross Baldick

Qixing Huang

Peter Stone

Evolutionary Neural Architecture Search for Deep Learning

by

Jason Zhi Liang,

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2018

This thesis is dedicated to my parents, who have supported my dreams of pursuing a doctorate degree, and to all the UTCS PhD students that are looking forward to graduation.

Acknowledgments

I wish to thank my research supervisor Risto Miikkulainen for his invaluable advice and feedback that helped guide my research over the past few years. It was due to Risto's graduate course that I was introduced to neural networks and evolutionary computation in the first place. Without his guidance, my dreams of becoming an artificial intelligence research scientist would not have been possible.

I will also like to acknowledge everyone in the Neural Network Research Group at UT Austin, especially Elliot Meyerson, for their helpful contributions and suggestions to help me further my research. Elliot collaborated with me on several projects and experiments and proved to be a very inspiring and thought provoking colleague. I would also give thanks to the LEAF team at Sentient Technologies, especially Dan Fink and Karl Mutch, for their assistance in building the infrastructure and framework needed for my research. Lastly, I like to thank my committee members, Peter Stone, Ross Baldick, and Qixing Huang, for their thoughtful suggestions and patience in reviewing my dissertation.

Evolutionary Neural Architecture Search for Deep Learning

Publication No. _____

Jason Zhi Liang, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Risto Miikkulainen

Deep neural networks (DNNs) have produced state-of-the-art results in many benchmarks and problem domains. However, the success of DNNs depends on the proper configuration of its architecture and hyperparameters. DNNs are often not used to their full potential because it is difficult to determine what architectures and hyperparameters should be used. While several approaches have been proposed, computational complexity of searching large design spaces makes them impractical for large modern DNNs.

This dissertation introduces an efficient evolutionary algorithm (EA) for simultaneous optimization of DNN architecture and hyperparameters. It builds upon extensive past research of evolutionary optimization of neural network structure. Various improvements to the core algorithm are introduced, including: (1) discovering DNN architectures of arbitrary complexity; (1) generating modular, repetitive modules commonly seen in state-of-the-art DNNs;

(3) extending to the multitask learning and multiobjective optimization domains; (4) maximizing performance and reducing wasted computation through asynchronous evaluations. Experimental results in image classification, image captioning, and multialphabet character recognition show that the approach is able to evolve networks that are competitive with or even exceed hand-designed networks. Thus, the method enables an automated and streamlined process to optimize DNN architectures for a given problem and can be widely applied to solve harder tasks.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **xii**

List of Figures **xiii**

Chapter 1. Introduction **1**
1.1 Motivation 2
1.2 Challenges 4
1.3 Approach 6
1.4 Dissertation Overview 9

Chapter 2. Background **11**
2.1 Deep Neural Networks and Deep Learning 11
2.1.1 Principles of Deep Learning 11
2.1.2 Common Deep Neural Network Architectures 13
2.1.3 Deep Multitask Learning 15
2.2 Evolutionary Algorithms 17
2.2.1 Neuroevolution 17
2.2.2 Neuroevolution of Augment Topologies 18
2.2.3 Asynchronous Evolutionary Algorithms 21
2.2.4 Evolutionary Bilevel Optimization 22
2.2.5 Multiobjective Evolutionary Algorithms 24
2.2.6 Evolutionary Novelty Search 26
2.3 Hyperparameter Optimization of Deep Neural Networks 28
2.4 Architecture Search for Deep Neural Networks 30

2.4.1 Reinforcement Learning Based Algorithms for Architecture Search . . . 30
2.4.2 Evolutionary Algorithms for Architecture Search 32
2.5 Conclusion . . . 33

Chapter 3. Evolution of Deep Neural Network Architectures 35
3.1 Motivation . . . 35
3.2 Extending NEAT to Evolve Deep Neural Networks . . . 36
3.2.1 Algorithm Description . . . 37
3.2.2 Massively Distributed Evaluation and Training of DNNs 41
3.3 Experimental Results in CIFAR-10 Image Classification Domain 42
3.3.1 CIFAR-10 Domain Overview . . . 42
3.3.2 Setup for CIFAR-10 Domain . . . 44
3.3.3 Results for CIFAR-10 Domain . . . 45
3.4 Conclusion . . . 46

Chapter 4. Coevolution of Modular Deep Neural Network Architectures 47
4.1 Motivation . . . 47
4.2 Coevolution of Blueprints and Modules . . . 49
4.2.1 Algorithm Overview . . . 49
4.2.2 Evolving Modular and Repetitive Structure . . . 52
4.3 Accelerating Coevolution with Asynchronous Evaluations . . . 53
4.3.1 Algorithm Overview . . . 53
4.3.2 Generic Asynchronous Evaluation Strategy . . . 55
4.3.3 Asynchronous Evaluation Strategy for CoDeepNEAT . . 56
4.4 Experimental Results in CIFAR-10 Image Classification Domain 57
4.4.1 Setup for CIFAR-10 Domain . . . 57
4.4.2 Results for CIFAR-10 Domain . . . 59
4.5 Experimental Results in MSCOCO Image Captioning Domain 60
4.5.1 MSCOCO Domain Overview . . . 61
4.5.2 Setup for MSCOCO Domain . . . 62
4.5.3 Results for MSCOCO Domain for CoDeepNEAT . . . 65

4.5.4 Results on MSCOCO domain for CoDeepNEAT-AES . . 67
4.6 Experimental Results in Wikidetox Comment Classification Domain . . . 69
4.6.1 Wikidetox Domain Overview . . . 69
4.6.2 Setup for the Wikidetox Domain . . . 70
4.6.3 Experimental Results in the Wikidetox Domain . . . 72
4.7 Real-World Use of DNNs Evolved with CoDeepNEAT . . . 75
4.8 Conclusion . . . 80

Chapter 5. Coevolution of Deep Neural Network Architectures for Multitask Learning 81
5.1 Motivation . . . 81
5.2 Soft-ordering Architecture . . . 83
5.3 Coevolution of Modules . . . 85
5.3.1 Algorithm Overview . . . 85
5.3.2 Weight Sharing Between Modules . . . 88
5.4 Coevolution of Modules and Shared Routing . . . 89
5.4.1 Algorithm Overview . . . 89
5.4.2 Random Population Initialization . . . 90
5.5 Experimental Results for Omniglot Multitask Learning Domain 91
5.5.1 Omniglot Domain Overview . . . 91
5.5.2 Setup for Omniglot Domain . . . 92
5.5.3 Results for Omniglot Domain . . . 94
5.6 Experimental Results for Chest X-Ray Multitask Learning Domain . . . 98
5.6.1 Chest X-ray Domain Overview . . . 98
5.6.2 Setup for Chest X-ray Domain . . . 99
5.6.3 Results for Chest X-ray Domain . . . 102
5.7 Conclusion . . . 104

Chapter 6. Multiobjective Coevolution of Deep Neural Network Architectures **107**
6.1 Motivation . . . 107
6.2 Multiobjective CoDeepNEAT . . . 109
6.2.1 Algorithm Overview . . . 109
6.2.2 Combining Multiobjective CoDeepNEAT with Novelty Search . . . 112
6.2.3 Experimental Results in MSCOCO Image Captioning Domain . . . 113
6.3 Multiobjective CMSR . . . 116
6.3.1 Using Multiobjective CMSR for Network Complexity Minimization . . . 116
6.3.2 Experimental Results in Chest X-ray domain . . . 118
6.4 Conclusion . . . 123

Chapter 7. Discussion and Future Work **124**
7.1 DeepNEAT and CoDeepNEAT . . . 124
7.1.1 Discussion . . . 125
7.1.2 Future Work . . . 127
7.2 CoDeepNEAT-AES . . . 140
7.3 CM and CMSR . . . 143
7.4 MCDN and MCMSR . . . 145
7.5 Conclusion . . . 147

Chapter 8. Conclusion **148**
8.1 Contributions . . . 149
8.2 Concluding Remarks . . . 151

Bibliography **153**

Vita **178**

List of Tables

3.1 Node and global hyperparameters evolved in the CIFAR-10 domain. They show just how large of a search space that DeepNEAT is exploring. 44

4.1 Summary of classification accuracy of best evolved networks from different approaches on CIFAR-10 domain. The networks evolved using DeepNEAT and CoDeepNEAT are highlighted in bold. The networks produced through architecture or hyperparameter search are labeled with an asterisk. Both CoDeepNEAT and DNGO outperform the manually designed Network-in-Network. 58

4.2 Node and global hyperparameters evolved for the image captioning case study. They show the size of the search space explored by CoDeepNEAT. 62

4.3 Summary of performance of different approaches on MSCOCO domain. The networks produced through architecture or hyperparameter search are labeled with an asterisk. The evolved network using CoDeepNEAT (highlighted in bold) improves over the hand-designed baselines and other architecture search methods. In particular, it was able to beat the Show and Tell network network by 5%. 66

5.1 Average validation and test accuracy over 20 tasks for each algorithm. CMSR performs the best as it combines both module and routing evolution. Pairwise t-tests show all differences are statistically significant with $p < 0.05$. 93

5.2 Summary of different search spaces; for normal and expanded, the input size is 224×224. For encoder, the input size is 28×28. 100

5.3 AUROC on test set for existing approaches that use hand-designed architectures and networks which are evolved using CMSR. CMSR combined with the hypercolumn approach results in an architecture that is competitive with the state-of-the-art. 103

List of Figures

2.1 A visualization of a number of DNN architectures that been explored so far by the deep learning research community [148]. This diversity suggests that architecture choice is important. . 15

2.2 Figure 2.2a shows how NEAT [141] mutates a chromosome (representing a neural network) by either incrementally adding a node (neuron) or a edge (weight) between two nodes. Figure 2.2b shows how two chromosomes perform crossover by swapping edges whose innovation numbers match. NEAT can be extended to neural architecture search by representing layers in a DNN as nodes. 19

2.3 A visualization of a set of solutions with respect to two objectives (network complexity and fitness) and the Pareto front (green line) where the Pareto optimal solutions reside. The Pareto front contain the best possible trade-offs between the two objectives. 25

3.1 Overview of the algorithm for DeepNEAT and how it evolves networks. The main difference between NEAT and DeepNEAT is that in DeepNEAT the chromosome represents DNN architectures at the layer level rather than at the neuron level. . . . 37

3.2 Figure 3.2a shows an example DeepNEAT chromosome while Figure 3.2b shows corresponding DNN architecture that is created from parsing the chromosome. Note the chromosome graph is represented as a list of nodes and edges and each node has its own set of evolvable hyperparameters. The chromosome also has a set of global hyperparameters that are relevant to the DNN as a whole. This representation allows the evolution of arbitrary DNN topologies. 38

3.3 Examples of the images from the 10 classes of CIFAR-10 dataset [73]. As a standard benchmark for testing DNNs, this dataset is a good way to evaluate the effectiveness of neural architectures discovered by DeepNEAT. 43

3.4 Visualization of the best network evolved by DeepNEAT on the CIFAR-10 domain. This architecture includes a lot of shortcut connections that lack of any regular, modular structure. The performance of this architecture is comparable to a similar hand-designed network [87]. 45

4.1 GoogLeNet [145], an example of a DNN with modular and repetitive structure. The inception module is shown on the left while the full network architecture is shown on the right (with the module circled in red). 48

4.2 A visualization of how CoDeepNEAT assembles networks for fitness evaluation. Modules and blueprints are assembled together into a network through replacement of blueprint nodes with corresponding modules. This approach allows evolving repetitive and deep structures seen in many recent DNNs. 50

4.3 Overview of the algorithm for CoDeepNEAT and how it uses coevolution to create assembled networks from separate blueprint and module populations. 51

4.4 Visualization of best assembled networks discovered by CoDeepNEAT at generations 1, 40, and 60. In the first generation, the networks are minimal and have no modules. However, by generation 40, the networks contain modules that are repeated at multiple locations (highlighted in red). 52

4.5 Overview of the GAES, a generic version of CoDeepNEAT-AES that can be applied to any parallel but synchronous EA. . . . 55

4.6 Overview of CoDeepNEAT-AES, an asynchronous extension of CoDeepNEAT that can take full advantage of a pool of workers for evaluations, made available through the completion service. 56

4.7 Top: High level visualization of the best network evolved by CoDeepNEAT for the CIFAR-10 domain. Node 1 is the input layer, while Node 2 is the output layer. The network has repetitive structure because its blueprint reuses same module in multiple places. Bottom: A more detailed visualization of the entire network with the locations of the modules highlighted in red. The use of modules allowed CoDeepNEAT to beat both DeepNEAT and a hand-designed architecture. 60

4.8 Some examples of the types of images in the MSCOCO dataset: (a) iconic object images, (b) iconic scene images, and (c) non-iconic images [22]. 61

4.9 Visualization of the best architecture found by evolution. Among the components in its unique structure are six LSTM layers, four summing merge layers, and several skip connections. A single module architecture (highlighted in red) consisting of two LSTM layers merged by a sum is repeated three times. There is a path from the input through dense layers to the output that bypasses all LSTM layers, providing the softmax with a more direct view of the current input. The power of the architecture seems to come from the many shortcut connections, which are unlikely to have been discovered by hand. 65

4.10 A plot of fitness versus time elapsed for synchronous CoDeepNEAT and CoDeepNEAT-AES. Each marker in the plot represents the fitness at a different generation. At any given time, CoDeepNEAT-AES was able to achieve much better fitness. . 68

4.11 A plot of fitness versus number of generations elapsed for synchronous CoDeepNEAT and CoDeepNEAT-AES. This result shows that both algorithms achieved the same fitness when compared by generations. However, the generations for CoDeepNEAT-AES were much shorter in duration. 68

4.12 A comparison of CoDeepNEAT against the networks discovered via several commercially available methods, including Kaggle, MSFT TLC, MOE, and Google AutoML. The Y-axis shows best fitness/accuracy achieved so far, while the X-axis shows the generations, total training time, and total amount of money spent on cloud compute. As the plot shows, CoDeepNEAT is gradually able to discover better networks, eventually finding one in the 40th generation that beats all other approaches. . . 72

4.13 A visualization of the best network discovered by CoDeepNEAT in the Wikidetox domain. As the network architecture shows, it is most optimal to use a combination of both simple and complex modules within the blueprint of the network. 74

4.14 An iconic image from an online magazine captioned by an evolved model. The model provides a suitably detailed description without any unnecessary context. 77

4.15 Results for captions generated by an evolved model for the online magazine images rated from 1 to 4, with 4=Correct, 3=Mostly Correct, 2=Mostly Incorrect, 1=Incorrect. Left: On iconic images, the model is able to get about one half correct; Right: On all images, the model gets about one fifth correct. The superior performance on iconic images shows that it is useful to build supplementary training sets for specific image types. 78

4.16 Top: Four good captions. The model is able to abstract about ambiguous images and even describe drawings, along with photos of objects in context. Bottom: Four bad captions. When it fails, the output of the model still contains some correct sense of the image. The results overall are promising and suggest that the model can be improved by including more difficult images in the training set. 79

5.1 The relationships of various MTL architectures described in this chapter. The soft-ordering method [99] is used as the starting point, extending it with CoDeepNEAT, leading to CM and CMSR on the bottom. 83

5.2 Example soft-ordering network with three shared layers [99]. Soft-ordering learns how to use the same layers in different locations by learning a tensor S of task-specific scaling parameters. S is learned jointly with the W_d, to allow flexible sharing across tasks and depths. This architecture enables the learning of layers that are used in different ways at different depths for different tasks. 85

5.3 High-level algorithm outlines of CM and CMSR, illustrating how they are similar and different from each other. In particular, the algorithms differ in whether or not the blueprint is evolved along with the modules (see line 7 in Algorithm 6). . . 86

5.4 Comparison of fitness (validation accuracy after partial training for 3000 iterations) over generations of single runs of CM and CMSR. Solid lines show the fitness of best assembled network and dotted line show the mean fitness. Both methods reach a similar fitness, but CMSR is slower to converge. 94

5.5 Comparison of fitness over generations of CM with disabling, enabling, and evolving module weight sharing. No sharing is better than forced sharing, but evolvable sharing outperforms them both, validating the approach. 95

5.6 Visualizations of the best networks evolved by CM (Figures 5.6a and 5.6b) and CMSR (Figures 5.6c and 5.6d) on the Omniglot domain. The module routing for CM is fixed but is evolvable in CMSR. The module routing (blueprint) evolved by CMSR contains many shortcut connections and could be a possible factor in CMSR's superior performance. Both methods evolve a mixture of both complex and simple module architectures. . 97

5.7 The top row shows negative examples of images from the Chest X-ray dataset where no disease is present. The bottom row show positive examples where the images do show a disease. . 98

5.8 Comparison of fitness over generations of CMSR with different search spaces as described in Table 5.2. Solid lines show the fitness of best assembled network and dotted lines shows the mean fitness. The hypercolumn search space is quickest to converge and reaches the best fitness. 102

5.9 Overview of the best architectures evolved for the expanded (Figures 5.9a and 5.9b) and hypercolumn (Figures 5.9c and 5.9d) search spaces. The best hypercolumn network is significantly simpler than the best expanded network. 105

6.1 Overview of lower and upper levels of MCDN perform ranking of individuals and also how the Pareto front is calculated from two objectives. 111

6.2 List of the hand-crafted features used to characterize the behavior of each evolved network for novelty search. The novelty score generated using the behavior metric is used as a secondary objective along with fitness. 113

6.3 Comparison of fitness over generations of MCDN (multiobjective) and CoDeepNEAT (single-objective) on the MSCOCO image captioning domain. Solid lines show the fitness of best assembled network and dotted lines shows the mean fitness. MCDN is able to converge at generation 15, 15 generations faster than CoDeepNEAT. 114

6.4 Visualizations of best networks evolved by MCDN (Figure 6.4a) and CoDeepNEAT (Figure 6.4b) on the image captioning domain. The modules in both networks are highlighted in red. Both networks are able to reach similar fitness after six epochs of training, but the network evolved by MCDN is significantly more complex and contains more novel module structures. . . 115

6.5 Comparison of fitness over generations of the single-objective CMSR and multiobjective MCMSR with the normal search space (Table 5.2). Solid lines show the fitness of best assembled network and dotted lines shows the mean fitness. Minimizing network complexity also seems to benefit the primary objective; MCMSR is able to achieve higher fitness and converge faster than CMSR. 118

6.6 Comparison of the single and multiobjective Pareto fronts for CMSR (green) and MCMSR (blue) respectively at various generations during evolution. The X-axis shows number of parameters (secondary objective) while Y-axis shows AUROC fitness (primary objective). The Pareto front for MCMSR consistently dominates over CMSR's Pareto front. In other words, MCMSR discovers trade-offs between complexity and performance that are always better than those found by CMSR. 120

6.7 Visualizations of networks with different complexity discovered by MCMSR. The performance of the significantly smaller 56K network (Figure 6.7a) is nearly as good as that of the larger 125K network (Figure 6.7b). The smaller network uses only two instances of the module architecture shown in Figure 6.7c while the larger network uses four instances of the same module. These two networks show that MCMSR is able to find good trade-offs between two conflicting objectives by clever usage of modules. 122

7.1 Comparison of the predictive performance of hand-crafted embeddings described in Figure 6.2 and learned embeddings using Deepwalk [111]. The histogram of error is shown on the left while the correlation between predicted and actual fitnesses is shown on the right. Deepwalk is able to predict the fitnesses of the networks with smaller mean absolute error (MAE) and with higher correlation to the actual fitnesses than the hand-designed embeddings. 133

7.2 Visualization of a sequence-to-sequence model. The model is composed of two LSTM layers [8]. The LSTM on the left is called the encoder and the LSTM on the right is called the decoder. This sequence-to-sequence network can be used to accurately predict the learning curves of DNNs. 135

7.3 Visualizations of the example predictions of a sequence-to-sequence model when given three epochs of training loss (green) and fitness (blue) as input. The solid line shows the ground-truth values while the break shows the first few epochs that were given as input to the model. The dotted line shows the predicted values of the model. The model is accurate and able to predict within 2% error of the ground-truth. 137

7.4 Histogram of time per generation for synchronous CoDeepNEAT and CoDeepNEAT-AES. The average time per generation for CoDeepNEAT-AES is significantly less than that for CoDeepNEAT. 140

7.5 Histogram of frequency of returned results over the course of a typical generation for both algorithms. CoDeepNEAT-AES wastes less time because the result results have a flat distribution compared to the Gaussian distribution for CoDeepNEAT. 141

7.6 Histogram comparing the delay between submission of individuals by the EA and when they are actually trained. The average delay time is longer for CoDeepNEAT-AES, but does not seem to negatively affect performance. 142

7.7 Visualization of the number of unique species created for both CoDeepNEAT (Figure 7.7a) and MCDN (Figure 7.7b) during evolution. Each species are represented as a unique color, with the X-axis showing the current generation. The increased number of species created by MCDN shows that it is able to maintain a more diverse population through using novelty as a secondary objective. 145

Chapter 1

Introduction

In industries such as manufacturing, finance and construction, automation has revolutionized how products are made and increased productivity by orders of magnitude. However, artificial intelligence (AI) and machine learning (ML) applications are still created by hand. While computational power used to be a bottleneck, the availability of cloud computing and extremely powerful GPUs has shifted the bottleneck to the research scientist; in other words, human time has become much more scarce than machine time. To make things worse, the number of people with the skills and qualifications to design AI and ML systems are highly limited and significant amount of effort is required to train them. If the creation and validation of AI applications can somehow be automated, it will lead to an explosion in productivity and significantly accelerate progress in AI technology as well. By making AI commonplace, it could lead to insights that will eventually make artificial intelligence (AGI) [40] possible. By creating a foundation to automate AI applications, this dissertation hopes to contribute to this vision.

1.1 Motivation

Machine learning and artificial intelligence have seen widespread growth in applications recently, driven by both improvements in computing power and dataset quality. In particular, in the past couple of years a special form of machine learning called deep learning has become possible. Deep learning [76] uses deep neural networks (DNN) to learn rich representations of high-dimensional data in either supervised or unsupervised manner. DNNs have exceeded the state-of-the-art in an variety of benchmarks that were previously dominated by other machine learning algorithms. These benchmarks include those in the computer vision, natural language processing, reinforcement learning, and speech speech recognition domains [25, 45, 52, 103].

One noticeable trend in deep learning is that state-of-the-art DNN are becoming more complex and their performance depends more on their architecture and choice of hyperparameters [20, 52, 109, 144]. Furthermore, a lot research in deep learning is focused on discovering of specialized architectures that excel in specific tasks. Due to a lack of theoretical understanding of DNNs, it is difficult to predict the performance of a DNN without empirically testing it on a benchmark task. There is much variation between DNN architectures (even for a single task) and so far, there is no guiding principle for deciding what is the right architecture is for a task. Finding the right architecture and hyperparameters is essentially reduced to a black-box optimization process. However, manual testing and evaluation of architectures and related hyperparameters is a tedious and time consuming task. Often the parameters

and choice of architecture are chosen based on history and convenience rather than solid principles.

Some attempts have been made at partial automation. The authors might tune a few hyperparameters or switch between several fixed architectures, but rarely optimize both the architecture and hyperparameters simultaneously. This approach is understandable since the search space is massive and existing methods do not scale as the the number of hyperparameters and architecture complexity increases. The standard and most widely used methods for hyperparameter optimization is grid search [149], which involves the discretization of hyperparameters into a fixed number of intervals and then exhaustingly searching through all possible combinations. Each combination is tested by training a DNN with those hyperparameters and evaluating its performance with respect to a metric on a benchmark dataset. While this method is simple and can be parallelized easily, its computational complexity grows exponentially with the number of hyperparameters, and becomes intractable once the number of hyperparameters exceeds four or five [68]. Grid search also does not address the question of what the optimal architecture of the DNN should be, which may be just as important as the choice of hyperparameters.

Thus, manual and grid search methods are insufficient for finding state-of-the-art DNN architectures for real world domains. Alternative, more automated methods should be considered. One key question, which always surrounds architecture search algorithms, is whether they can discover solutions that exceed the performance of hand-designed ones. This dissertation aims to

show that DNN architectures and hyperparameters that are optimized using an automated algorithm can match or even exceed the performance of hand-designed and hand-tuned networks. This process is done through an empirical comparison of performance with the latest published results on multiple benchmarks that are commonly used in the deep learning community.

1.2 Challenges

Given the drawbacks of existing methods in finding the right architecture for a task, an automatic method for searching architectures that can scale with the complexity of the search space and number of hyperparameters becomes all the more important. However, there are many challenges in creating such an automated system.

One major issue is that there is no gradient information for such search. Normally, in the training process for DNNs, the gradient can be computed for the network parameters with respect to the input in order to minimize the output error [53]. This computation is possible because the weights in a neural network are mathematically related to the input vector. However, no such relationship exists between the network's architecture and the input. Without any gradient information, the search for the right DNN becomes a black-box optimization problem where only the value of the objective function (i.e. performance of DNN on a task) is known [64]. Furthermore, the search space for network architectures is non-Euclidean and the arrangement of layers in a DNN can take on any arbitrary graph structure with arbitrary numbers

of both discrete and real-valued hyperparameters. Any practical automated method for optimizing network structure must be able to handle such a large and complex search space efficiently.

For many problems and benchmarks, it is important to discover good network architectures that can perform well on not just one, but several tasks. In other words, network architecture search must be applicable to multitask learning (MTL) as well [19]. Unfortunately, architecture search becomes more complicated as the right architecture for one task might not be the right architecture for another task. Human-designed networks have shown that it is possible to create architectures where each task is treated differently and different layers in the network are used to process each task uniquely [99]. However, adapting such techniques to an automated method for architecture search is hard and remains an open challenge.

Like multitask learning, other types of problems require exploring different trade-offs in multiple metrics in the networks whose structure is being optimized. For example, in mobile applications, it is important to find network architectures that have as few parameters as possible (e.g. fitting in the memory of a smartphone), but also the best possible performance on a benchmark. Because the size of a network and its performance are often inversely correlated, it is important for the architecture search algorithm be able to balance both objectives and discover a range of solutions that offer the best trade-off between the two objectives.

In order for an automated architecture search algorithm to optimize

DNNs, it must have to search through hundreds, if not thousands of network topologies. Unfortunately, DNNs are computationally very intensive to train and require specialized hardware such as graphics processing units (GPUs) to train. Even with the right hardware, complex network architectures such as GoogLeNet [145] require weeks to train. Unsurprising, the total number of GPU hours required is extremely high (up to tens of thousands) for a single experimental run and attempting to train thousands of networks on a single machine is thus impractical. Luckily, the rise of cloud computing [1] has recently made hundreds of GPU equipped machines to be made available at a click of a button. The challenge of distributing the training of networks around the machines and making sure that the all available machines are being utilized fully with none being idle still remains though.

This dissertation will solve the challenges listed above by using an evolutionary approach. Evolutionary algorithms (EA) are black-box optimization algorithms that can efficiently search through high-dimensional, large, and complex search spaces [30, 80, 131]. Since there already exists EAs designed for efficient exploration of graph topologies, EAs present a promising starting point for improving neural architecture search.

1.3 Approach

In this dissertation a novel algorithm called DeepNEAT is proposed, where an existing neuroevolution method called Neuroevolution of Augmenting Topologies (NEAT) [141] is extended to optimize the topology and hyper-

parameters of a DNN simultaneously. NEAT is unique among other EAs in that it can start from a minimal architecture and explore new ones through incremental complexification. Furthermore, there are heuristics within NEAT which are designed to make the network complexification process as efficient as possible. Thus NEAT serves as an excellent foundation for constructing an EA for neural architecture search.

While NEAT is powerful, it cannot search through very large search spaces where networks can have hundreds of layers. Thus, an extension to NEAT is developed that increases the diversity of networks that it can evolve and allows it to generate repetitive and modular topologies commonly seen in state-of-the-art DNNs. This version of NEAT, named CoDeepNEAT, utilizes coevolution to evolve two separate populations, one of blueprints and the other of modules. The two populations are combined to generate a much larger assembled network. In this approach, when CoDeepNEAT evolves the blueprint and modules incrementally, it results in a much larger change in the structure of the assembled network. As a result, very deep and modular network architectures can be evaluated much earlier during evolution and those components (modules) in the architecture that work will be preserved for future generations.

To tackle architecture search for multitask learning, CoDeepNEAT is adapted to take advantage of a recent innovation in MTL called soft-ordering [99]. When combined with soft-ordering, CoDeepNEAT can evolve network architectures that reuse modules of layers in different ways for each task. As a

result, the evolved networks become much more flexible and process each task in the best suitable way compared to conventional DNNs designed for MTL. Similarly, to optimize problem domains where there are two or more objectives that might also be conflicting, CoDeepNEAT is modified to take advantage of the vast amounts of work done with multiobjective evolutionary optimization [168]. Inspired by NSGA-II [31], an approach that ranks solutions not by a single metric, but by a Pareto front constructed from multiple objectives is developed.

Because it depends on training the candidate DNNs, evolutionary optimization of network architectures require massive amounts of computational resources. To speed up evolution in CoDeepNEAT, the evaluation of each candidate during evolution is done on a separate and dedicated machine that is equipped with GPU. While parallelization can help, each network requires a different amount of time to train and there still lies the issue of machines becoming idle at the end of each generation. To solve this problem, CoDeepNEAT is modified to make use of a new asynchronous evaluation strategy. This new strategy adapts and improves upon strategies that have been applied successfully to asynchronous but non-distributed EAs [138] and minimizes the time the worker machines spend idling and not evaluating networks.

One key question always surrounding automated search algorithms is whether they can discover solutions that exceed the performance of hand-designed ones. The novel approaches to architecture search in this dissertation are tested in real-world domains, with the purpose of showing optimized ar-

chitectures can match or exceed the performance of hand-designed networks. This dissertation includes such empirical comparisons with published results in the following types of benchmark tasks: (1) image classification (assignment of a label that most fit the content of the image), (2) image captioning (generation of human readable text that best describes the image), and (3) multitask image classification (assignment of one or more labels to an image). The results of these comparisons validate the effectiveness of the algorithms introduced by this dissertation.

1.4 Dissertation Overview

The rest of this dissertation is organized as follows: (1) First, the foundations of CoDeepNEAT and relevant related work are discussed in more detail, including topics and areas such as deep learning, evolutionary computation, and previous work done in neural architecture search. (2) Second, an overview of DeepNEAT is given, including how it is modified and extended from NEAT. Experimental results in the CIFAR-10 image classification domain are reported. (3) Third, an improvement to DeepNEAT called CoDeepNEAT is described in detail, including how it applies the principles of coevolution to architecture search. Experimental results in the CIFAR-10 domain and MSCOCO image captioning domain are provided. (4) Fourth, CoDeepNEAT is extended to multitask learning and evaluated experimentally in the Omniglot character recognition and Chest X-ray image classification domains. (5) Fifth, CoDeepNEAT is extended to the multiobjective optimization and

evaluated in the MSCOCO and Chest X-ray domains. (6) Sixth, a discussion session is included to provide a higher level analysis for the experimental results of the algorithms and also to propose interesting directions for future work.

Chapter 2

Background

The work in this dissertation, evolutionary neural architecture search, builds upon two major areas of research within the machine learning and artificial intelligence community: deep learning and evolutionary algorithms. Thus, this chapter reviews foundations and related work from these two areas of research that are relevant to the topic of this dissertation. First, an introduction to deep learning and deep neural networks is given. Second, evolutionary optimization of neural networks is reviewed. Third, related work in neural architecture and hyperparameter search is surveyed.

2.1 Deep Neural Networks and Deep Learning

This section will give an introduction deep learning, in particular, the principles of deep learning, commonly used architectures, and the application of DNNs to multitask learning.

2.1.1 Principles of Deep Learning

Deep learning is a relatively new and fast developing field of machine learning. It uses multiple processing layers to learn representations of data

with multiple levels of abstraction [76]. These processing layers are commonly implemented using neural networks, which are mathematical functions that process an input vector through a combination of linear operations (matrix multiplication) and non-linear operations (element-wise transformation using an activation function such as logistic sigmoid, hyperbolic tangent, or rectified linear unit). Neural networks have desirable properties such as full differentiability, graceful degradation, and resistance to noise/errors in inputs and model parameters. Most importantly, a neural network can approximate any arbitrarily complex function if the network has enough hidden neurons [27]. Because they are differentiable, neural networks can also be efficiently trained through the backpropagation algorithm, a form of stochastic gradient descent (SGD) [124].

Deep learning has its roots in neural network research dating back to the early 2000's, when researchers discovered that by repeatably stacking multiple neural network layers (thus making the network deeper), performance, instead of dropping due to overfitting and parameterization, actually improved. There has been many attempts at explaining this counter-intuitive behavior but a commonly accepted explanation is that the combination and thus representational power of features learned by neural networks increases exponentially with both the hidden layer size and depth [76]. While shallow neural networks typically have at most two layers, the layer count for deep neural networks (DNNs) often can exceed 100 [51,57]. Initially, DNNs were pretrained greedily layer by layer in an unsupervised fashion and then fine tuned on a supervised

training set, such as with the stacked denoising autoencoder [149] or restricted boltzmann machine architectures. However, recent advances in DNN research showed that deep networks can be trained directly with SGD if the weights are initialized from a suitable distribution and the learning rate is reasonably low.

2.1.2 Common Deep Neural Network Architectures

Among the various types of DNN architectures, convolutional neural networks (CNN) are perhaps the most popular kind of architecture in applications. CNNs are distinguished from regular neural networks by their weight connectivity pattern, which is shared, sparse, and locally connected. As a result, CNNs are especially well suited for processing spatially structured tasks commonly seen in computer vision and image processing. The first CNN architecture to gain popularity was AlexNet [73], which set a new record on the ImageNet large-scale image classification competition. AlexNet demonstrated the power of stacking multiple convolutional layers and of using linear rectified units (ReLU) as activation functions. Two other innovative architectures introduced for image classification included: (1) GoogleNet [145], which made use of multiple output channels and a novel modular layer structure that is repeated stacked together, and (2) ResNet [51], which used shortcut, additive transformations known as residual connections. CNNs are also useful for the image embedding vectors that the second to last output layer generates. In tasks such as object detection [39, 121] or image captioning [150], these em-

beddings are fed as input into another DNN that performs a different function than the original image classification CNN.

Recurrent networks are a type of DNN architecture that is popular in tasks that require processing sequences. Sequence learning is found in reinforcement learning, natural language processing, language modeling, speech recognition, and time-series data. Recurrent networks are characterized by backward connections that feed the output of a layer in the previous timestep to the input in the current timestep. They are trained using the backpropagation-through-time algorithm [153], where the networks are unrolled to form an feedforward network (with each layer sharing the same weights) and then updated with regular backpropagation. One particular type of recurrent layer called the long short-term memory (LSTM) [55] solves the vanishing gradient problem commonly seen in recurrent networks, where the gradient update becomes negligible as the distance between the layer being updated and the output layer increases. LSTM layers avoid this problem by having various soft gating functions that can allow or impede the flow of gradient information through the recurrent connections selectively. In the Penn Treebank [97] language modeling task, LSTMs were able to beat state-of-the-art, non deep learning approaches such as hidden Markov and n-gram models [100]. Similarly LSTMs set recording-breaking results on the image captioning task (MSCOCO dataset) that involves generating sentences from image embeddings [150].

Besides the popular DNN architectures mentioned above, there are many other novel DNN architectures designed for more niche applications.

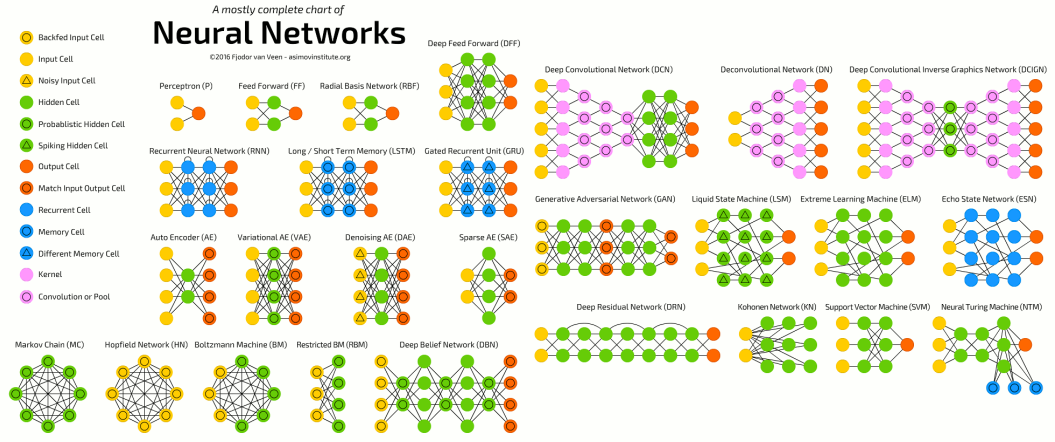


Figure 2.1: A visualization of a number of DNN architectures that been explored so far by the deep learning research community [148]. This diversity suggests that architecture choice is important.

A comprehensive visualization of all the different types of DNNs is shown in Figure 2.1. The diversity in network topologies suggests that network structure indeed matters and an automated method for discovering them could be useful.

2.1.3 Deep Multitask Learning

Multitask Learning (MTL) [19] exploits relationships across problems to increase overall performance. The underlying idea is that if multiple tasks are related, the optimal models for those tasks will be related as well. In the convex optimization setting, this idea has been implemented via various regularization penalties on shared parameter matrices [14, 37, 66, 74]. Evolutionary methods have also had success in MTL, especially in sequential decision-

making domains [60, 63, 67, 125, 135].

Deep MTL extended these ideas to domains where deep learning thrives, including vision [17, 65, 92, 102, 115, 120, 161, 167], speech [58, 59, 65, 129, 157], natural language processing [24, 33, 49, 65, 90, 95, 166], and reinforcement learning [28, 62, 146]. The key decision in constructing a deep multitask network is how parameters such as convolutional kernels or weight matrices are shared across tasks. Designing a deep neural network for a single task is already a high-dimensional open-ended optimization problem; having to design a network for multiple tasks *and* deciding how these networks share parameters grows this search space combinatorially. Most existing approaches draw from the deep learning perspective that each task has an underlying feature hierarchy, and tasks are related through an *a priori* alignment of their respective hierarchies. These methods have been reviewed in more detail in previous work [99, 123]. Another existing approach adapts network structure by learning task hierarchies, though it still assumes this strong hierarchical feature alignment [92].

DNN architecture choice plays an very important role in MTL because there are many ways to tie multiple tasks together. The best network architectures are large and complex, and have become very hard for humans to design, thus demonstrating the necessity of automated methods for optimizing network topologies in the MTL domain.

2.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) and their applications to neural networks are described in this section. Background that is necessary to understand the algorithms designed in this dissertation include neuroevolution, asynchronous evolution, bilevel optimization, multiobjective optimization, and novelty search for evolution.

2.2.1 Neuroevolution

In neuroevolution, evolutionary algorithms (EAs) are used to optimize a neural network with respect to its performance on a task [78]. Neuroevolution techniques have been applied successfully to sequential decision tasks for three decades [38, 79, 104, 162]. In such tasks there is no gradient available, so instead of gradient descent, evolution is used to optimize the weights of the neural network. Neuroevolution is a good approach in particular to POMDP (partially observable Markov decision process) problems: It is possible to evolve recurrent connections to allow disambiguation of hidden states.

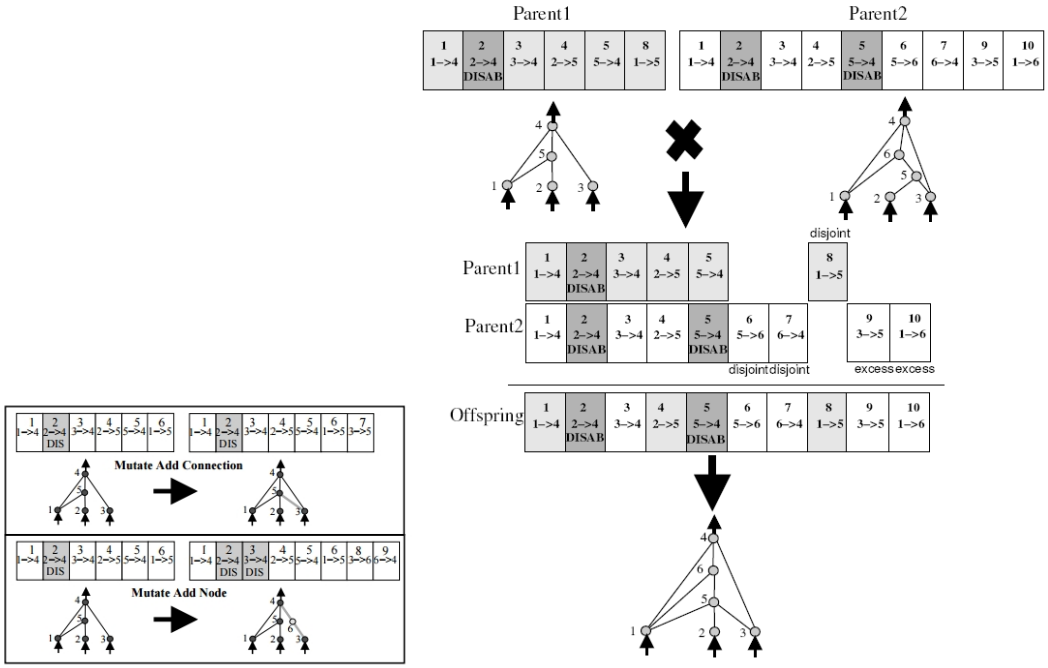
Neural network weights can be optimized using various evolutionary techniques. Genetic algorithms are a natural choice because crossover is a good match with neural networks: They recombine parts of existing neural networks to find better ones. CMA-ES [47], a technique for continuous optimization, works well on optimizing the weights as well because it can capture interactions between them. Other approaches such as SANE, ESP, and CoSyNE evolve partial neural networks and combine them into fully functional

networks [41, 42, 106]. Further, techniques such as Cellular Encoding [46] and NEAT [141] have been developed to evolve the topology of the neural network, which is particularly effective in determining the required recurrence. Neuroevolution techniques work well in many tasks in control, robotics, constructing intelligent agents for games, and artificial life [78]. However, because of the large number of weights to be optimized, they are generally limited to relatively small networks.

Evolution can also be combined with gradient-based learning, making it possible to utilize much larger networks. These methods are still usually applied to sequential decision tasks, but gradients from a related task (such as prediction of the next sensory inputs) are used to help search. Much of the work is based on utilizing the Baldwin effect, where learning only affects the selection [54]. Computationally, it is possible to utilize Lamarckian evolution as well, i.e. encode the learned weight changes back into the genome [46]. However, care must be taken to maintain diversity so that evolution can continue to innovate when all individuals are learning similar behavior.

2.2.2 Neuroevolution of Augment Topologies

Neuroevolution of Augment Topologies (NEAT) is an evolutionary algorithm that is especially designed for the evolution of neural networks. Unlike conventional EAs that can only optimize fixed vector representations, NEAT can evolve the topology and weights of a neural network simultaneously. It does so by representing networks with a graph-like chromosome or individual



(a) Mutation (b) Crossover

Figure 2.2: Figure 2.2a shows how NEAT [141] mutates a chromosome (representing a neural network) by either incrementally adding a node (neuron) or a edge (weight) between two nodes. Figure 2.2b shows how two chromosomes perform crossover by swapping edges whose innovation numbers match. NEAT can be extended to neural architecture search by representing layers in a DNN as nodes.

that is made of two lists of edges and nodes. Besides unique genetic encoding, NEAT also has the following features that allow it to evolve topology efficiently: (1) Edges have innovation numbers to keep track of mutations to the structure of the network, thus allowing fast alignment during crossover. (2) Networks are grouped according to their similarity to each other into separate

subpopulations called species. Selection, mutation and crossover only happen within a species and the size of each species is updated according the overfall fitness of the species. Speciation ensures that newly created network topologies are protected from competition with existing ones. (3) Networks start up from a minimal structure (composed of only input and output neurons) and undergo complexification through mutations that add either edges or nodes to the network. This heuristic reduces the search space and adds a regularization effect that encourages simple solutions. Figure 2.2 shows example NEAT chromosomes and how the mutation and crossover operations are performed.

NEAT works well in many tasks in control, robotics, constructing intelligent agents for games, and artificial life [78], especially where the domain is continuous and supervised learning is not possible. For example in the double pole-balancing control task, NEAT discovered a single and elegant solution that intelligently made use of recurrent connections [140]. In an approach similar to the work in this dissertation, but in the reinforcement learning domain, NEAT was used to evolve the structure of neural networks whose weights were then trained with SGD [154].

However, NEAT is not without its limitations: The networks evolved by NEAT have a messy irregular structure and it is difficult to scale NEAT to a large number of inputs and outputs. Some modifications and variants of NEAT attempted to address these deficiencies. In HyperNEAT [139], NEAT is used to evolve a compositional pattern producing network or CPPN. This CPPN is then use to generate the weight or connectivity pattern of a much large network

that then is evaluated on the task at hand. By using an indirect encoding, HyperNEAT allows NEAT to generate extremely large networks that exhibit regular, repetitive weight connectivity patterns. HyperNEAT was shown to be effective for solving complex reinforcement learning tasks such as Atari game playing, where the state space is extremely high [50]. Due to its flexibility in representing graphs and ease of extensibility, it is natural to use NEAT as a starting point for optimizing the architecture of DNNs.

2.2.3 Asynchronous Evolutionary Algorithms

Asynchronous master-slave evolutionary algorithms have been around since the early 1990s, and have been occasionally used by practitioners [32, 94, 116]. However, little work is done analyzing the behavior and benefits of such systems [69, 128, 165]. Such methods have recently become more relevant when parameter tuning for large simulations has become more common.

A popular asynchronous EA for evolving both the weights and topology of neural networks is rtNEAT [138]. In that system, a population of neural networks are evaluated asynchronously one at a time. Each neural network is tested in a video game, and its fitness measured over a set time period. At the end of the period, it is taken out of the game; if it evaluated well, it is mutated and crossed over with another candidate to create an offspring that is then tested in the game. In this manner, evolution and evaluation are happening continually at the same time. The goal of rtNEAT is to make replacement less disruptive for the player; it does not provide any performance advantage

because each individual is still evaluated in the same environment. Asynchrony can help speed up neural architecture search by reducing the average idle time of workers in a distributed EA, as this dissertation will later show.

2.2.4 Evolutionary Bilevel Optimization

Neural architecture search and hyperparameter tuning are closely related to bilevel (or multilevel) optimization techniques [133]. The general idea of bilevel optimization is to use an evolutionary optimization process at a high level to optimize the parameters of a low-level evolutionary optimization process. This two level scheme is similar to neural architecture search where evolution is used only to optimize the design of the neural network at the higher level and gradient descent is used to optimize or train the weights of the network at the lower-level.

More formally, bilevel optimization [26] describes a special class of optimization problems where there are two levels of optimization tasks: an upper-level optimization task with parameters p_u and objective function F_u, and a lower-level optimization task with parameters p_l and objective function F_l. The goal is find a p_u that allows p_l to be optimally solved:

$$\begin{aligned} &\underset{p_u}{\text{maximize}}\ F_u(p_u) = E[F_l(p_l)|p_u] \\ &\text{subject to } p_l = O_l(p_u), \end{aligned} \tag{2.1}$$

where $E[F_l(p_l)|p_u]$ is the expected performance of the lower-level solution p_l obtained by lower-level optimization algorithm O_l with p_u as its parameters.

The maximization is done by a separate upper-level optimization algorithm O_u. Given that bilevel optimization involves nested optimization, it is also very computationally complex and usually requires large scale distributed computation for solving real world problems.

Consider for instance the problem of controlling a helicopter through aileron, elevator, rudder, and rotor inputs. This is a challenging benchmark from the 2000s for which various reinforcement learning approaches have been developed [13,15,108]. One of the most successful ones is single-level neuroevolution, where the helicopter is controlled by a neural network that is evolved through genetic algorithms [72]. The eight parameters of the neuroevolution method (such mutation and crossover rate, probability, and amount and population and elite size) are optimized by hand. It would be difficult to include more parameters in this process because the parameters interact nonlinearly. A large part of the parameter space thus remains unexplored in the single-level neuroevolution approach. However, a bilevel approach, where a high-level evolutionary process is employed to optimize these parameters, can search this space more effectively [85]. With bilevel evolution, the number of parameters optimized could be extended to 15, which resulted in significantly better performance. In this manner, evolution was harnessed to optimize a system design that was too complex to be optimized by hand.

While bilevel optimization has many applications, the particular application that is popular in the deep learning community is hyperparameter tuning. In such tuning, the goal is for O_u to find the optimal parameters

p_u such that it maximizes the performance of O_l. Because parameter tuning is usually a black-box optimization task involving non-differentiable objective functions, O_u is usually implemented as an EA. In past work on hyperparameter optimization for neuroevolution [86], bilevel optimization was shown to be an effective way of tuning neuroevolution for complex control tasks such as double-pole balancing and helicopter hovering. Furthermore, having more hyperparameters to tune for O_l seems to lead to higher optimized performance. Surrogate optimization, where $E[F_l(p_l)|p_u]$ is estimated through a regression function (instead of through the actual task at hand), is an efficient way to reduce the computational complexity of bilevel optimization. These findings are especially important in the context of evolving DNNs since they often contain many hyperparameters and require a long time for training and performance evaluation.

2.2.5 Multiobjective Evolutionary Algorithms

Multiobjective optimization is an extension of the single-objective case: Instead of having only one objective or fitness function to optimize, there are two or more such functions. A multiobjective maximization problem [5, 168] is defined as:

$$\underset{x}{\text{maximize}}(F_1(x), F_2(x), ..., F_i(x)) \tag{2.2}$$

where $x \in X$ is a vector being optimized, $F_i(x)$ are the objective functions and X is the set of all feasible solutions for x. A multiobjective minimization

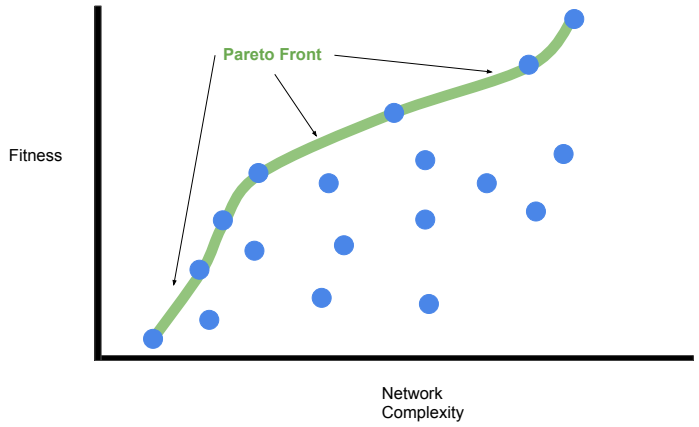


Figure 2.3: A visualization of a set of solutions with respect to two objectives (network complexity and fitness) and the Pareto front (green line) where the Pareto optimal solutions reside. The Pareto front contain the best possible trade-offs between the two objectives.

problem can be defined by simply replacing the maximization of $F_i(x)$ with minimization. Because there does not usually exist a solution x that can maximize or minimize all of the objective functions, it is necessary to find solutions which are Pareto optimal. Such solutions are those which cannot improve with respect to one objective without reducing another objective. In other words, there does not exist another solution y that dominates a Pareto optimal solution x, where domination is defined as:

$$\begin{aligned} F_j(y) &\geq F_j(x) \quad \forall j \in 1, 2, ..., i \\ F_j(y) &> F_j(x) \quad \exists j \in 1, 2, ..., i \end{aligned} \tag{2.3}$$

The set of Pareto optimal solutions are also called a Pareto front [168] and the target of most multiobjective optimization algorithms is to discover such

a Pareto front as efficiently as possible. A visualization of an example Pareto front is shown in Figure 2.3.

Many single-objective evolutionary algorithms have been adapted to multiobjective search, including algorithms that make use of population-based heuristics such as particle swarm optimization [105], differential evolution [159], or probability distributions as in CMA-ES [61]. A popular, widely used multiobjective EA is NSGA-II [31], which utilizes a crowding-distance heuristic along with genetic operators to discover Pareto fronts efficiently. NSGA-II was integrated into NEAT [126, 127] to evolve game-agent controllers that are capable of complex and multimodal behavior. In addition, NSGA-II was combined with other heuristics such as novelty search (Section 2.2.6) [82, 130] to encourage more diversity during evolution and help the EA escape local optima in the fitness landscape. Evolutionary multiobjective optimization is highly applicable to neural architecture search and as this dissertation will show, can discover architectures with good trade-offs between network size and performance.

2.2.6 Evolutionary Novelty Search

Because EAs are often utilized to optimize non-convex objective functions, overcoming deception or local optima in the search space is extremely important. Such a problem requires avoiding premature converge of the population towards a single target and maintaining diverse individuals within the population. While multiobjective optimization based only on fitness (per-

formance with respect to some task) can help with diversity, it does have limitations and often fails in real-world applications. Novelty search [80] is a powerful technique to allow evolution to escape deceptive traps in the fitness landscape. In novelty search, a behavior metric or vector that is typically uncorrelated to the fitness, is defined over the individuals in the population and is calculated during evaluation of the individual. After evaluation, behavior metrics are added to an archive, and the distance (typically euclidean) is calculated between the behavior metric of each individual and the closest one in the archive. The most novel individuals with the greatest distance from the others are then selected to be persisted into the next generation.

Novelty search has been used with success in neuroevolution and reinforcement learning domains [80, 81, 83, 98]. The behavior metric is often derived from interactions or behavior of the neural network controller with the environment. For example in the maze domain, where an agent must navigate obstacles to reach an end position, the behavior is the end position of the agent after a fix amount of time. One surprising finding is that novelty alone could be used during evolution and often can outperform fitness based search, especially in deceptive domains [81]. Novelty search can also be used as a secondary objective along with fitness as the primary objective [82, 130] for multiobjective optimization. As this dissertation will show later, novelty search and fitness information can be combined with multiobjective search to accelerate convergence during evolution.

2.3 Hyperparameter Optimization of Deep Neural Networks

Efficient hyperparameter optimization of DNNs have become increasing important part of deep learning research as DNNs grow in complexity, take longer to train, and are more sensitive to architecture choice. DNN hyperparameter search is classified as a type black-box optimization problem, where the objective function to be optimized (relation between hyperparameters and DNN performance on some benchmark) has no analytical form and derivatives cannot be calculated. Instead, the function can only be evaluated at points that are explicitly specified. As a result, gradient-based methods cannot be used for finding the optimum for black-box objective functions. Instead, families of diverse algorithms have been developed for this purpose.

The simplest form of hyperparameter optimization is exhaustive grid search, where points in hyperparameter space are sampled uniformly at regular intervals [149]. One of the earliest improvements from grid search is randomized hyperparameter search [16] where the points are randomly chosen from a uniform distribution. Randomized search yielded significant improvements over grid search for certain types of DNNs due to the fact that many hyperparameters of these DNNs are insensitive to tuning, thus the effective dimensionality of the search space is lower than expected. Grid search cannot sample efficiently from this lower dimension space, but random search is able to. Like grid search, random search can be trivial parallelized over a distributed cluster of machines to reduce the overall running time.

Although random search proved to effective for certain networks, it suffers when all the hyperparameters are crucial to the performance of the DNN and must be optimized to very particular values. For networks that suffer from such characteristics, Bayesian optimization using Gaussian processes [136] proved to a feasible alternative instead. The strength of Bayesian optimization is that it requires relatively few function evaluations and works reasonably well on multimodal, non-separable, and noisy functions where there are several local optima. It works by first creating a probability distribution of functions (also known as Gaussian process) that best fits the objective function. As more points of the objective function are evaluated, the distribution becomes more well defined and the Bayesian optimization algorithm becomes more certain of the portions of hyperparameter space that should be examined and used to sample new points for evaluation. The main weakness of Bayesian optimization is that it is computational expensive and scales cubically with then number of evaluated points. DNGO [137] tried to address this issue by replacing Gaussian processes with Bayesian neural networks that exhibited linear scaling. Another downside of Bayesian optimization is that empirically, it has been observed to perform poorly when the number of hyperparameters to be optimized is moderately high [91].

EAs have been another class of algorithms which are widely used for black-box optimization of complex, multimodal functions. They rely on biological inspired mechanisms to iteratively improve upon a population of candidate solutions to the objective function. One particular EA that has been success-

fully applied to DNN hyperparameter tuning is CMA-ES [91]. In CMA-ES, a Gaussian distribution for the best individuals in the population is estimated and used to generate/sample the population for the next generation. Furthermore, it has mechanisms for adaptively controlling the step-size and the direction that the population will move. CMA-ES has been shown to perform well in many real-world high dimensional optimization problems and in experiments performed by Loshchilov et al. [91], CMA-ES has been shown to outperform Bayesian optimization on tuning the parameters of a convolutional neural network.

However, hyperparameter search is fundamentally constrained by the fact that the underlying DNN architecture might not be optimal. More advanced methods, like the ones in this dissertation, can jointly optimize the hyperparameters and architecture of the network.

2.4 Architecture Search for Deep Neural Networks

This section will summarize related techniques and algorithms that have been used by the deep learning community to optimize both the hyperparameters and the architecture of DNNs.

2.4.1 Reinforcement Learning Based Algorithms for Architecture Search

The main drawback of the hyperparameter optimization methods described in Section 2.3 is that they cannot modify the topology of the DNN and

they are thus dependent on a suitable base architecture being supplied beforehand. However, there has been research recently into reinforcement learning based architecture (RL) search algorithms. One solution is to use an recurrent neural network (LSTM) controller to generate a sequence of layers that begin from the input and end at the output of a DNN [169]. The LSTM controller is trained through RL, in particular, a gradient-based policy search algorithm called REINFORCE [155]. The architecture space explored by this approach is sufficiently large and is even capable of generating skip-connections between layers. On a popular image classification benchmarks such as CIFAR-10 and ImageNet, such an approach achieved performance within 1-2 percentage points of the state-of-the-art, and on a language modeling benchmark, it achieved state-of-the-art performance at the time [169, 170].

However, the space of architectures that the algorithm can search through is still relatively limited. For example in the RL approach [169], the architecture of the optimized network must have either a linear or tree-like core structure; arbitrary graph topologies are outside the search space. Thus it is still up to the user to define an appropriate search space beforehand for the algorithm to use as a starting point. The number of hyperparameters that can be optimized for each layer are also limited. Furthermore the computations are extremely heavy; to generate the final best network, many thousands of candidate architectures have to be evaluated and trained, which requires hundreds of thousands of GPU hours.

Consequently, recent work on RL based methods [88, 112] focused on

improving the efficiency of architecture search and maximizing the performance of the discovered network architectures when given limited computational budget. One approach [88] made use of surrogate optimization while another approach [112] utilized parameter sharing between different candidate architectures. Such approaches discovered architectures with excellent results in image classification domains and repeatedly advanced the state-of-the-art in the language modeling domain.

Besides reinforcement learning, architecture search can also be done using evolutionary black-box optimization (as this dissertation will show later). Evolutionary methods have certain advantages over RL based algorithms that will be described in more detail in Section 2.4.2.

2.4.2 Evolutionary Algorithms for Architecture Search

One particularly promising direction for architecture search is the application of evolutionary algorithms (EAs). Evolutionary methods are well suited for this kind of problems because they are black-box optimization algorithms that do not depend on gradient information. Some of these approaches use a modified version of NEAT [118], an EA for neuron-level neuroevolution [141], for searching network topologies. Others rely on genetic programming [117, 142] or hierarchical evolution [89]. There is some very recent work on multiobjective evolutionary architecture search [34,93], where the goal is to optimize both the performance and training time/complexity of the network.

The main advantage of EAs over RL methods is that they can optimize

over much larger search spaces. For instance, approaches based on NEAT [118] can evolve arbitrary graph topologies for the the network architecture. Most importantly, hierarchical evolutionary methods [89], can search over very large spaces efficiently and evolve complex architectures quickly from a minimal starting point. As a result, the performance of evolutionary approaches match or exceed that of reinforcement learning methods. For example, the current state-of-the-art results on image recognition benchmarks such as CIFAR-10 and ImageNet are achieved by an evolutionary approach [119]. This dissertation will present a novel EA that is capable of evolving networks with arbitrary topology in a hierarchical manner.

Evolutionary architecture share with RL the drawback of computational complexity. Large populations of candidate network architectures must be evaluated at each generation. The state-of-the-art results mentioned above are only possible if thousands of machines equipped with GPUs are available. Recent work has been done to make evolutionary architecture search more efficient [36, 142]. Reductions in computational cost are accomplished by using better mutation operations or genetic encodings, thus reducing the population size required.

2.5 Conclusion

This chapter gave an overview of the foundations and related work for neural architecture search, a relatively new area of research. The principles behind NEAT, the foundation of the approach proposed in this dissertation, were

discussed and the main two approaches to network architecture optimization, reinforcement learning and evolutionary algorithms were summarized. The pros and cons of each approach, especially the strengths behind evolutionary architecture search, were reviewed. This dissertation will draw inspiration from many of the topics presented in this chapter to create a novel architecture search algorithm. Furthermore, other ideas such as hierarchical search and asynchrony will serve as the basis for more advanced algorithms presented in the later chapters.

Chapter 3

Evolution of Deep Neural Network Architectures

This chapter introduces a NEAT-based based neuroevolution algorithm name DeepNEAT for evolving DNN architectures. It builds on foundations in neuroevolution and NEAT, reviewed in Section 2.2. In this chapter the motivation behind using a NEAT-based approach to architecture search is first explained. DeepNEAT is described in detail, especially how it modifies and extends NEAT. In addition, DeepNEAT is applied to evolve DNN architectures in the CIFAR-10 image classification domain and experimental results which are competitive with similar hand-designed networks are presented.

3.1 Motivation

Many approaches to network architecture search are powerful when given a limited search space, but the topologies of the network must follow certain constraints or patterns. They are less powerful when the search space is expanded to cover arbitrary graphs or if there are too many hyperparameters to optimizer. For example, in the reinforcement learning (RL) based algorithm proposed by Zoph et al. [169], the networks that are discovered

must have either a linear or tree-like core structure depending on the domain. Furthermore the number of hyperparameters that could be optimized for each layer is limited.

As black-box optimization methods, evolutionary algorithms (EA) are especially suited for architecture search, where no gradient information with respect to the network structure is available. Compared to RL based methods, evolutionary methods [118, 119] have recently shown promising results when optimizing over large search spaces where networks have arbitrary structure. Given that NEAT is a powerful EA capable of evolving arbitrary structure and has discovered state-of-the-art networks in many domains such as robotics and agents for games, it is a promising starting point for designing an EA for performing architecture search. NEAT already has built in mechanisms such as speciation and incremental complexification that allows it to efficiently search over large amounts of networks with highly diverse structure. Although NEAT evolves networks at the neuron level and abstracts each neuron into a node in a graph, the abstraction can be easily modified so that each node represents a layer in the network instead, thus extending it to neural architecture search. Lastly, NEAT is a mature, widely used algorithm and many open source implementations of it are available as a base for building DeepNEAT.

3.2 Extending NEAT to Evolve Deep Neural Networks

DeepNEAT is a most immediate extension of the standard neural network topology-evolution method NEAT to DNN. This section will describe

Algorithm 1 DeepNEAT

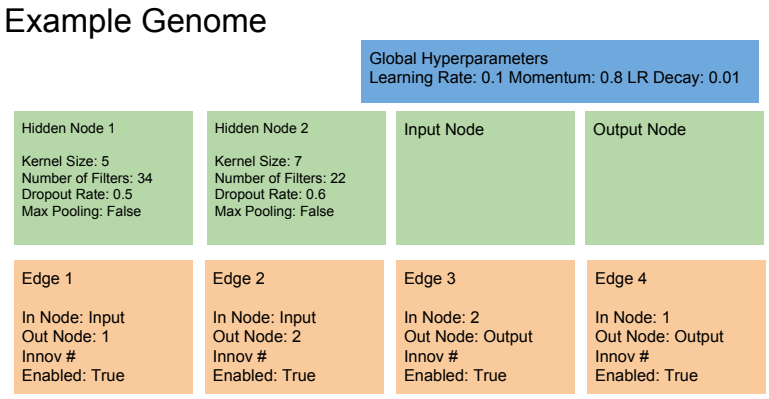
1. **Initialize** population of chromosomes P and empty species S
2. **For each** generation:
   3. **Group** each chromosome P_i into a species S_i based on similarity
   4. **For each** species S_i
      5. **Truncate** S_i and remove worst chromosomes
      6. **Generate** new chromosomes by performing following operations:
         7. **Tournament selection** from remaining chromosomes
         8. **Mutation** of selected chromosomes
         9. **Crossover** of selected chromosomes
      10. **Add** new chromosomes to S_i
   11. **Evaluate** the fitness of each P_i

Figure 3.1: Overview of the algorithm for DeepNEAT and how it evolves networks. The main difference between NEAT and DeepNEAT is that in DeepNEAT the chromosome represents DNN architectures at the layer level rather than at the neuron level.

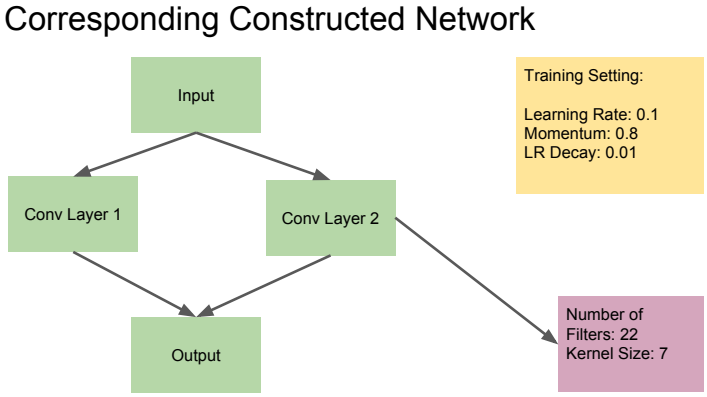
how DeepNEAT is modified to make use of the NEAT's neural-level topology evolution to evolve architectures of DNNs instead. In addition, a method for improving the performance of DeepNEAT and parallelizing the evaluation of networks is presented.

3.2.1 Algorithm Description

DeepNEAT follows the same fundamental process as NEAT: First, a population of chromosomes of minimal complexity is created. Each chromosome is represented as a graph and is also referred to as an individual. Over generations, structure (i.e. nodes and edges) is added to the graph incremen-



(a) Example DeepNEAT chromosome.



(b) Corresponding DNN architecture.

Figure 3.2: Figure 3.2a shows an example DeepNEAT chromosome while Figure 3.2b shows corresponding DNN architecture that is created from parsing the chromosome. Note the chromosome graph is represented as a list of nodes and edges and each node has its own set of evolvable hyperparameters. The chromosome also has a set of global hyperparameters that are relevant to the DNN as a whole. This representation allows the evolution of arbitrary DNN topologies.

tally through mutation. As in NEAT, the mutation involves randomly adding a node or a connection between two nodes. During crossover, historical markings are used to determine how genes of two chromosomes can be lined up and which nodes can be randomly crossed over. The population is divided into species (i.e. subpopulations) based on a similarity metric. Each species grows proportionally to its fitness and evolution occurs separately in each species. Algorithm 1 gives an overview of how DeepNEAT evolves networks; overall the process is very similar to NEAT from a high-level perspective. For a visualization of how mutation and crossover occurs in DeepNEAT, please refer to Figure 2.2.

DeepNEAT differs from NEAT in that each node in the chromosome no longer represents a neuron, but a layer in a DNN. Each node contains a table of real and binary valued hyperparameters that are mutated through uniform Gaussian distribution and random bit-flipping, respectively. These hyperparameters determine the type of layer (such as convolutional, fully connected, or recurrent) and the properties of that layer (such as number of neurons, kernel size, and activation function). The edges in the chromosome are no longer marked with weights; instead they simply indicate how the nodes (layers) are connected.

To construct a DNN from a DeepNEAT chromosome, one simply needs to traverse the chromosome graph, replacing each node with the corresponding layer. The chromosome also contains a set of global hyperparameters applicable to the entire network (such as learning rate, training algorithm, and

data preprocessing). Figure 3.2a visualizes a simple example DeepNEAT chromosome while Figure 3.2b shows the structure of the corresponding network constructed from the chromosome.

When arbitrary connectivity is allowed between layers, additional complexity is required. If the current layer has multiple parent layers, a merge layer must be applied to the parents in order to ensure that the parent layer's output is the same size as the current layer's input. Typically, this adjustment is done through a concatenation or element-wise sum operation. If the parent layers have mismatched output sizes, all of the parent layers must be downsampled to parent layer with the smallest output size. The specific method for downsampling is domain dependent. For example, in domains such as image classification, a max-pooling layer is inserted after specific parent convolutional layers while in other domains such as image captioning that use recurrent layers, a fully connected bottleneck layer will serve this function.

During fitness evaluation, each chromosome is converted into a DNN. These DNNs are then trained for a fixed number of epochs. After training, a metric that indicates the network's performance is returned back to DeepNEAT and assigned as fitness to the corresponding chromosome in the population. While DeepNEAT can be used to evolve DNNs, the resulting structures are often complex and unprincipled. They contrast with typical DNN architectures that utilize repetition of basic components. In Chapter 4, DeepNEAT is extended to allow for evolution of modular, repetitive networks.

3.2.2 Massively Distributed Evaluation and Training of DNNs

One of the main challenges involved in using DeepNEAT to evolve the architecture and hyperparameters of DNNs is the sheer raw computational power required to evaluate and train the networks in the population every generation. For example, each network in domain such as image classification requires anywhere from one to three days of training to reach full convergence. With a typical evolution run of 50 generations and a population size of 100, hundreds of thousands of GPU hours would have to be spent. Running DeepNEAT on a single computer would take over 13 years for just one experiment! As a compromise, the time spent training the networks is capped by limiting the number of epochs, but even then, the amount of computation required is still immense. In order to keep the running time of DeepNEAT tractable, the evaluation of individuals in the population every generation is parallelized and distributed over hundreds of worker machines, each equipped with a dedicated GPU. For most of the experiments described in this dissertation, the worker machines are GPU equipped EC2 instances running on Amazon AWS, a widely used platform for cloud computing [1].

To this end, a software framework called the completion service (it is now part of a larger open source package called StudioML [9]) was created for allowing DeepNEAT to communicate and send jobs to the worker machines. Completion service allows a server node running DeepNEAT to train the assembled networks on worker GPU nodes running on Amazon EC2. First, the server node submits networks ready for fitness evaluation to the completion

service. They are pushed onto a queue (buffer) and each available worker node pulls a single network from the queue to train. After training is finished, fitness is calculated for the network and results are immediately returned back to the server. The results are returned one at a time and without any order guarantee through a separate return queue. By using the completion service to parallelize evaluations, thousands of candidate networks are trained in a matter of days, thus making architecture search tractable.

3.3 Experimental Results in CIFAR-10 Image Classification Domain

In this section, DeepNEAT is applied to evolve networks in the CIFAR-10 image classification domain. Experimental results and visualizations of the best evolved networks are reported.

3.3.1 CIFAR-10 Domain Overview

In this experiment, CoDeepNEAT was used to evolve the topology of a convolutional neural network (CNN) to maximize its classification performance on the CIFAR-10 dataset [73], a widely used image classification benchmark in deep learning research. The dataset consists of 50,000 training images and 10,000 testing images. The images consist of 32x32 RGB color pixels and belong to one of 10 classes of commonly seen animals and objects. Examples of images from each class are shown in Figure 3.3. For comparison purposes, the neural network layer types was restricted to those used by DNGO [137]. DNGO

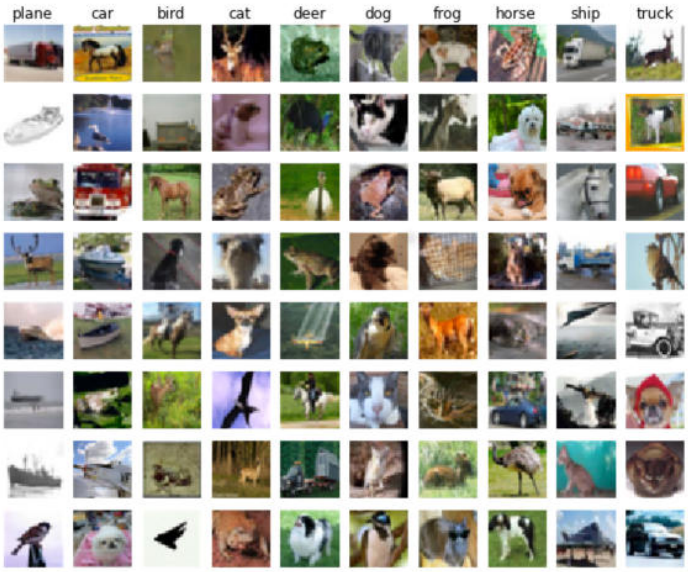


Figure 3.3: Examples of the images from the 10 classes of CIFAR-10 dataset [73]. As a standard benchmark for testing DNNs, this dataset is a good way to evaluate the effectiveness of neural architectures discovered by DeepNEAT.

uses Bayesian optimization for hyperparameter tuning and achieved state-of-the-art results for hyperparameter optimization in CIFAR-10 when this experiment was originally performed. Following the methodology in DNGO, data augmentation was also used and consisted of the following: Convert the images from RGB to HSV color space, add random perturbations, distortions, and crops, and convert them back to RGB color space.

Node Hyperparameter	**Range**
Number of Filters	$[32, 256]$
Dropout Rate	$[0, 0.7]$
Initial Weight Scaling	$[0, 2.0]$
Kernel Size	$\{1, 3\}$
Max Pooling	{True, False}
Global Hyperparameter	**Range**
Learning Rate	$[0.0001, 0.1]$
Momentum	$[0.68, 0.99]$
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}
Nesterov Accelerated Gradient	{True, False}

Table 3.1: Node and global hyperparameters evolved in the CIFAR-10 domain. They show just how large of a search space that DeepNEAT is exploring.

3.3.2 Setup for CIFAR-10 Domain

DeepNEAT is initialized with a population of 100 chromosomes. The search space and the list of hyperparameters evolved by DeepNEAT is described in Table 3.1. Its hyperparameters determine the various properties of the layer and whether additional max-pooling or dropout layers are attached. In addition, a set of global hyperparameters are evolved for the assembled network. During fitness evaluation, the 50,000 images in the CIFAR-10 dataset are split into a training set of 42,500 samples and a validation set of 7,500 samples. Each network is trained for eight epochs on the training set due to the size and complexity of the dataset. The validation set is then used to determine

classification accuracy, i.e. the fitness of the network. After 60 generations of evolution, the best network in the population is returned. Furthermore, as another way to keep the running time of the experiment reasonable, the evaluation of CNNs every generation are parallelized over 100 AWS EC2 workers, each equipped with a GPU for accelerating DNN training.

3.3.3 Results for CIFAR-10 Domain

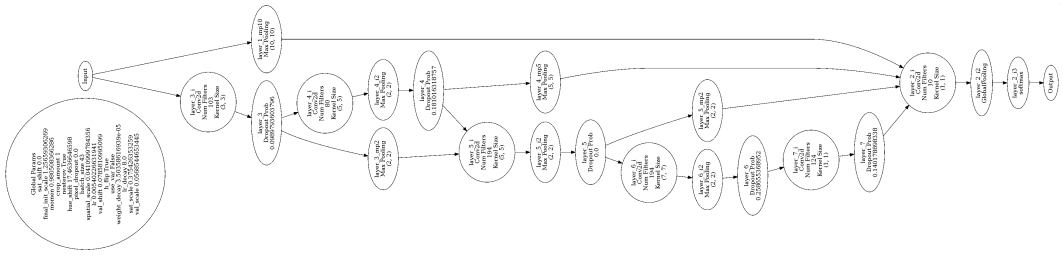


Figure 3.4: Visualization of the best network evolved by DeepNEAT on the CIFAR-10 domain. This architecture includes a lot of shortcut connections that lack of any regular, modular structure. The performance of this architecture is comparable to a similar hand-designed network [87].

After evolution was complete, the best network was trained on all 50,000 training images for 300 epochs, and the classification error on the test set was measured. The network was able to achieve 8.9% error, which is comparable to hand-designed networks with roughly the same number of parameters and similar layer types, such as the Network-in-Network architecture [87]. This result is not as good as the network (also with similar parameter count and layer types) optimized by DNGO [137], thus showing that there is still potential to improve DeepNEAT. For a comparison of DeepNEAT against other

approaches in the CIFAR-10 domain, please refer to Table 4.1 in Chapter 4.

Interestingly, because only limited training could be done during evolution, the best network evolved by DeepNEAT was optimized to train very fast. The evolved network is able to reach 20% test error after just 8 epochs and requires 80 epochs to converge. This is much faster than the networks optimized using other hyperparameter search methods such as DNGO [137], which requires 30 epochs to reach 20% test error and over 200 epochs to converge. The fast training of the network evolved by DeepNEAT is probably due to the numerous shortcut connections between layers at various depths within the network. Figure 3.4 shows a visualization of the best evolved network and its structure.

3.4 Conclusion

This chapter presented a novel EA called DeepNEAT that is capable of optimizing both the hyperparameters and architecture of a DNN. The approach is based on an existing and powerful neuroevolution algorithm called NEAT and makes use of mechanisms such as incremental complexity and speciation to efficient search for network architectures. Due to the computational complexity of training DNNs, a method for parallelizing the evaluation of the evolved networks was introduced. Results are presented in the CIFAR-10 domain, where DeepNEAT is capable of discovering networks that are competitive with hand-designed architectures.

Chapter 4

Coevolution of Modular Deep Neural Network Architectures

CoDeepNEAT, an extension of DeepNEAT that uses coevolution, is introduced in this chapter. It builds upon DeepNEAT, NEAT, and neuroevolution, which are reviewed in Chapter 3 and Section 2.2. First, the motivation behind coevolution of DNN architecture is reviewed. Second, the algorithms for CoDeepNEAT and an asynchronous variant of CoDeepNEAT are described. Third, experimental results are presented for the CIFAR-10 image classification domain, the MSCOCO image captioning domain, and the Wikidetox comment classification domain. They show that CoDeepNEAT can discover architectures competitive with state-of-the-art hand-designed networks.

4.1 Motivation

Many of the most successful DNNs, such as GoogLeNet and ResNet, are composed of modules that are repeated multiple times [51, 145]. These modules often themselves have complicated structure with branching and merging of various layers. For example, GoogLeNet is composed of stacking several inception modules on top of each other (Figure 4.1). Inception module's spe-

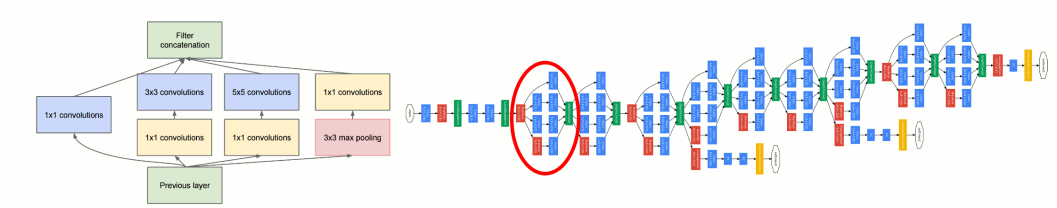


Figure 4.1: GoogLeNet [145], an example of a DNN with modular and repetitive structure. The inception module is shown on the left while the full network architecture is shown on the right (with the module circled in red).

cial architecture and bottleneck convolutional layers are believed to be largely responsible for the powerful performance of the network. Similarly, ResNet is able to improve performance simply by stacking a simple module many times to create a deeper network.

Although DeepNEAT can generate arbitrary network graphs, the stochastic nature of evolution means that it is highly unlikely for DeepNEAT to consistently evolve such modular structure. Furthermore, state-of-the-art networks like GoogleLeNet often have hundreds of layers and layer depths easily exceeding 30-40. Such complex architecture search spaces cannot be efficiently explored by DeepNEAT alone since it has to start from a minimal topology and slowly increase complexity through mutations.

Inspired by these observations and the limitations of DeepNEAT, an improved version of the EA called Coevolution DeepNEAT (CoDeepNEAT), is proposed. Coevolution is a commonly used technique in evolutionary computation to generate more interesting behavior during evaluation by combining simpler components together. It has been used with success in many

domains, including function optimization [113], modeling predator-prey relationships [163], and being combined with genetic programming for subroutine optimization [160]. The specific coevolutionary mechanism in CoDeepNEAT is inspired mainly by Hierarchical SANE [107] but is also influenced by component-evolution approaches of ESP [44] and CoSyNE [43]. Both CoDeepNEAT and SANE differ from conventional neuroevolution in that they do not evolve entire networks. Instead both approaches evolve components that are then assembled into complete networks for fitness evaluation. This fitness is then attributed back to the respective components and the components are evolved independently. In the case of SANE, the components are neurons, while in CoDeepNEAT, they are the DNN modules.

4.2 Coevolution of Blueprints and Modules

This section will explain in depth how DeepNEAT is modified to take advantage of coevolution and more sophisticated DNN architectures are evolved.

4.2.1 Algorithm Overview

As summarized in Algorithm 2, two populations of modules and blueprints are evolved separately, using the same methods as described in Chapter 3 for DeepNEAT. The blueprint chromosome (also known as an individual) is a graph where each node contains a pointer to a particular module species. In turn, each module chromosome is a graph that represents a small DNN. During fitness evaluation, the modules and blueprints are combined to create a

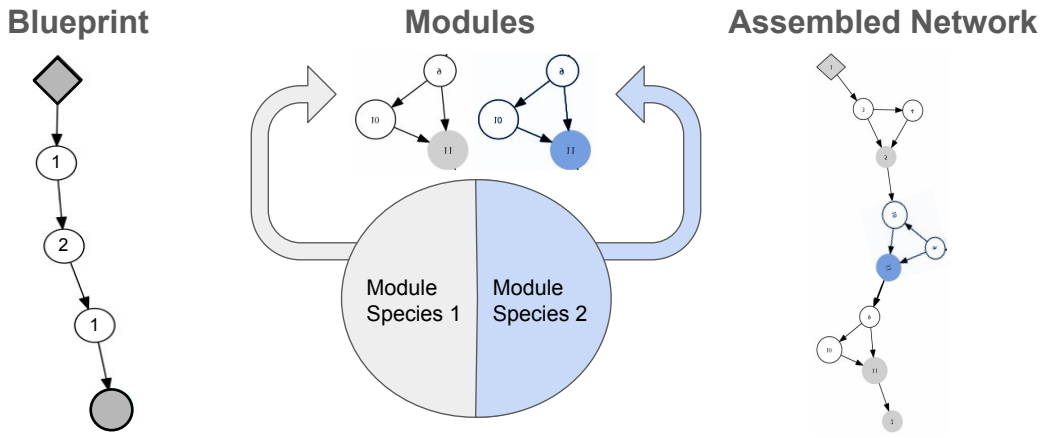


Figure 4.2: A visualization of how CoDeepNEAT assembles networks for fitness evaluation. Modules and blueprints are assembled together into a network through replacement of blueprint nodes with corresponding modules. This approach allows evolving repetitive and deep structures seen in many recent DNNs.

large assembled network. For each blueprint chromosome, each node in the blueprint's graph is replaced with a module chosen randomly from the species to which that node points. If multiple blueprint nodes point to the same module species, then the same module is used in all of them. After the nodes in the blueprint have been replaced, the individual is converted into a DNN in the same manner as DeepNEAT. The entire process for assembling the network is visualized in Figure 4.2.

The assembled networks are evaluated the a manner similar to DeepNEAT, but the fitnesses of the assembled networks are attributed back to blueprints and modules as the average fitness of all the assembled networks containing that blueprint or module. This fitness assignment scheme reduces

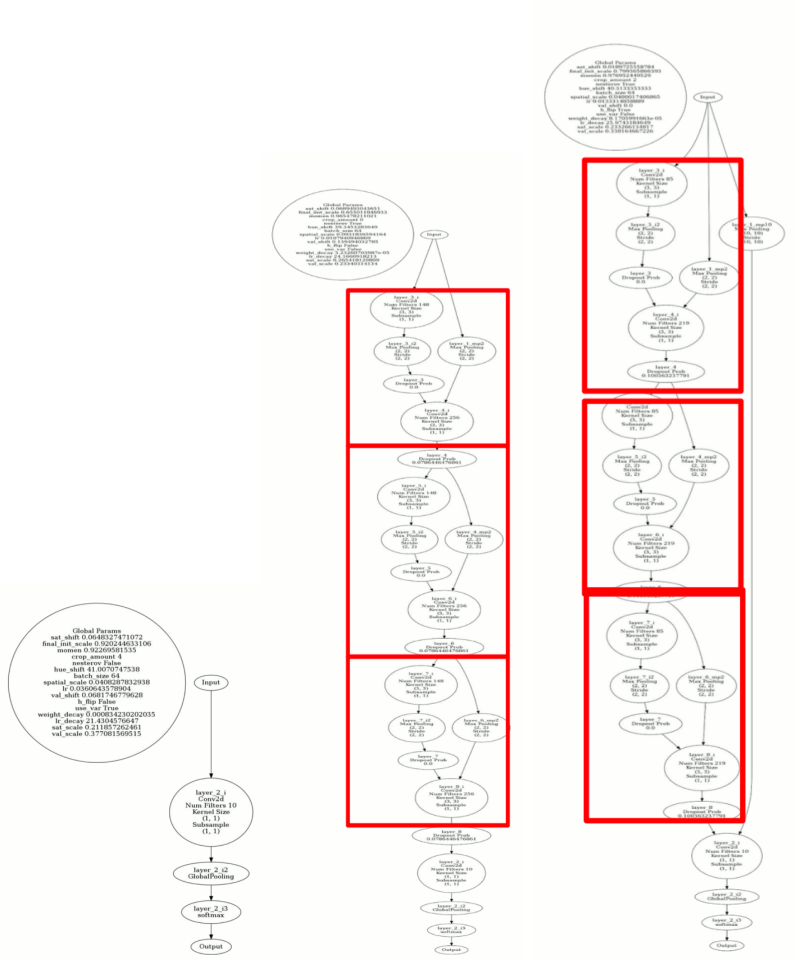
Algorithm 2 CoDeepNEAT

1. **Given** population of modules/blueprints
2. **For each** blueprint B_i during every generation:
   3. **For each** node N_j in B_i
      4. **Choose** randomly from module species that N_j points to
      5. **Replace** N_j with randomly chosen module M_j
   6. **When** all nodes in B_i are replaced, convert B_i to assembled network N_i
7. **Evaluate** fitnesses of the assembled networks N
8. **For each** network N_i
   9. **Attribute** fitness of N_i to its component blueprint B_i and modules M_j
10. **Evolve** blueprint and module population with DeepNEAT

Figure 4.3: Overview of the algorithm for CoDeepNEAT and how it uses coevolution to create assembled networks from separate blueprint and module populations.

evaluation noise and allows blueprints or modules to be preserved into the next generation even if they maybe included into an occasionally poorly performing network. There is also an alternative option in CoDeepNEAT that assigns the minimum or maximum fitness of the assembled networks to their components, however preliminary experiments have shown that using the mean fitness works best overall.

4.2.2 Evolving Modular and Repetitive Structure



(a) Generation 1 (b) Generation 40 (c) Generation 60

Figure 4.4: Visualization of best assembled networks discovered by CoDeepNEAT at generations 1, 40, and 60. In the first generation, the networks are minimal and have no modules. However, by generation 40, the networks contain modules that are repeated at multiple locations (highlighted in red).

CoDeepNEAT can evolve repetitive modular structure efficiently. Furthermore, because small mutations in the modules and blueprints often lead to large changes in the assembled network structure, CoDeepNEAT can explore more diverse and deeper architectures than DeepNEAT. An example application to the CIFAR-10 domain is presented next. Figure 4.4 shows the best network topologies discovered at generation 1, 40, and 60 of an example CoDeepNEAT run in the CIFAR-10 benchmark. Initially the network topology is minimal (Figure 4.4a). However as the generations progress, the networks eventually become not only deeper, but also show modular, repetitive structure (Figures 4.4b and 4.4c).

4.3 Accelerating Coevolution with Asynchronous Evaluations

Another optimization to the performance of CoDeepDEAT, asynchronous evaluation strategy (CoDeepNEAT-AES), is introduced below.

4.3.1 Algorithm Overview

A key problem that CoDeepNEAT-AES aims to solve is the inefficiency of synchronous evaluation when running an EA in a parallel, distributed environment. This problem is especially relevant to CoDeepNEAT since there is high variance in the complexity and the evaluation times of the assembled networks. As a result, evolution almost certainly becomes bottlenecked on waiting for the slowest individuals to be evaluated. CoDeepNEAT-AES will

try to alleviate this problem through two observations: (1) If there is a constant supply (buffer) of individuals to be readily evaluated, the worker nodes will have optimal throughput and minimal idle time because they are able to pull new individuals from the buffer immediately after evaluating the existing ones. (2) Server idle time can be minimized if evolution proceeds to the next generation immediately once a small fraction of the total number of individuals in the buffer have returned. In other words, it is no longer necessary to wait for the slowest individuals and CoDeepNEAT can evolve the next generation's population once there a sufficient number of fitness evaluations.

At any point in time, a queue (buffer) of K individuals ready to be evaluated are maintained. As computational resources become available, candidates are sent to them from this queue for evaluation. As soon as a predetermined batch size of M evaluations of individuals finish (where $M << K$), a new population of M individuals is generated from the returned individuals. This population is then submitted for evaluation in order to maintain a constant number of individuals on the buffer. In this manner, all available computational resources are used at all times. On the other hand, the process is no longer strictly generational, since individuals from several different generations may be evaluated in parallel.

Since the number of individuals in the buffer (K) greatly exceeds the number of individuals used to evolve the next generation (M), it is not scalable to have the EA keep track of all the individuals that are in the buffer and being evaluated on the worker nodes. The solution to this problem is simple: shift

Algorithm 3 GAES

1. **Given** large initial population with K individuals
2. **For each** generation:
   3. **Submit** every individual for evaluation with the worker nodes
   4. **Wait** for M individuals with fitnesses to return ($M * D = K$, $D > 1$)
   5. **Replace** existing population with returned individuals
   6. **Evolve** next generation's population of M individuals normally

Figure 4.5: Overview of the GAES, a generic version of CoDeepNEAT-AES that can be applied to any parallel but synchronous EA.

the burden of bookkeeping to the workers machines performing the evaluation of the DNNs. After the server places all the individuals it wishes to evaluate in the buffer, the server no longer keeps track of them. Instead, the workers return back to the server both the fitness values and the corresponding individuals together. In the case of CoDeepNEAT, the completion service provides both the buffer for which the server submits individuals to and a way for the worker nodes to return their results to the server. CoDeepNEAT then updates or overwrites the current population state with these newly returned individuals and their fitnesses.

4.3.2 Generic Asynchronous Evaluation Strategy

While this section is mainly about how asynchrony can improve the performance of CoDeepNEAT, it is interesting to note that such an evaluation strategy can be easily added onto any parallel, synchronous EA. Algorithm 3 describes a simple generic version of CoDeepNEAT-AES called GAES that has

Algorithm 4 CodeepNEAT-AES

1. **Given** initial blueprint/module populations
2. **For each** generation:
   3. **Generate** K assembled networks from blueprint/modules components
   4. **Submit** the networks along with components to completion service for evaluation
   5. **Wait** for M networks with fitnesses to return ($M * D = K$, $D > 1$)
   6. **Assign** blueprint/modules the fitnesses of their corresponding returned networks
   7. **Separate** blueprints/modules from the returned networks and filter out duplicates
   8. **Replace** existing assembled networks with returned assembled networks
   9. **Merge** returned blueprints/modules into existing blueprint/module populations
   10. **Evolve** the blueprint/module populations like in CoDeepNEAT

Figure 4.6: Overview of CoDeepNEAT-AES, an asynchronous extension of CoDeepNEAT that can take full advantage of a pool of workers for evaluations, made available through the completion service.

few assumptions regarding the underlying EA framework and adds no additional computational burden. K is the initial population size, M is the number of results to wait for (as well as population size for subsequent generations), and D is a hyperparameter which controls the ratio between K and M.

4.3.3 Asynchronous Evaluation Strategy for CoDeepNEAT

Algorithm 4 describes how CoDeepNEAT is modified to support asynchronous evaluations when evolving the topologies of DNNs. One main difference between CoDeepNEAT-AES and GAES is that there is no longer a persistent population of individuals. Instead, two populations of blueprints and modules are assembled together every generation to create a temporary population of networks. As a result, after getting results back from the comple-

tion service, CoDeepNEAT-AES not only updates the temporary population, but also the blueprint and module populations as well.

The blueprint and module populations are updated in two steps: (1) Determine the set of blueprints and modules that have returned from the worker nodes but also still exist in the populations. Replace the fitnesses of these already existing blueprints/modules with the latest updated fitnesses. (2) For the rest of the blueprints/modules, add them to the populations. By not discarding already existing individuals in the populations, it helps deal with degenerate cases in CoDeepNEAT-AES where blueprints do not point to any modules and thus as a result, no modules are returned along with the assembled networks. Furthermore, enlarging the pool of individuals used to generate the next generation's population enhances diversity.

4.4 Experimental Results in CIFAR-10 Image Classification Domain

This section will present results on evolving DNNs with CoDeepNEAT in the CIFAR-10 domain. Section 3.3.1 gives an overview of the CIFAR-10 domain.

4.4.1 Setup for CIFAR-10 Domain

CoDeepNEAT was initialized with populations of 25 blueprints and 45 modules. From these two populations, 100 CNNs are assembled for fitness evaluation in every generation. The search space was the same as in Deep-

Model	Test Error (%)
Network-in-Network [87]	8.8
DNGO* [137]	6.4
DeepNEAT*	**8.9**
CoDeepNEAT*	**7.3**

Table 4.1: Summary of classification accuracy of best evolved networks from different approaches on CIFAR-10 domain. The networks evolved using DeepNEAT and CoDeepNEAT are highlighted in bold. The networks produced through architecture or hyperparameter search are labeled with an asterisk. Both CoDeepNEAT and DNGO outperform the manually designed Network-in-Network.

NEAT (Table 3.1), where each node in the module represented a convolutional layer. The node's hyperparameters determined the various properties of the layer and whether additional max-pooling or dropout layers were added. In addition, a set of global hyperparameters were evolved for the assembled network. As with DeepNEAT, the 50,000 images were split into a training set of 42,500 samples and a validation set of 7,500 samples during fitness evaluation. Since training a DNN is computationally expensive, each network was trained for eight epochs on the training set. The validation set was then used to determine classification accuracy, i.e. the fitness of the network. After 72 generations of evolution, the best network in the population was returned. Like with DeepNEAT, parallelization was required to keep the evaluation time for CoDeepNEAT reasonable. The evaluation of DNNs in every generation were distributed over 100 GPU equipped EC2 instances.

4.4.2 Results for CIFAR-10 Domain

After evolution was complete, the best network was trained on all 50,000 training images for 300 epochs, and the classification error on the test set was measured. Table 4.1 compares CoDeepNEAT with DeepNEAT, DNGO (Bayesian optimization of hyperparameters) [137], Network-in-Network (a popular hand-designed architecture) [87]. All of the approaches were comparable since they produced networks with similar number of parameters and layer types. It is also important to note that DNGO had state-of-the-art results for hyperparameter optimization in CIFAR-10 when this experiment was run. The best network CoDeepNEAT discovered has 7.3% error, which is better than DeepNEAT and is within a single percentage of the 6.4% error for DNGO This is impressive considering that CoDeepNEAT was optimizing both the hyperparameters and the architecture of the network and the search space explored by CoDeepNEAT was far larger. Both CoDeepNEAT and DNGO outperformed the hand-designed Network-in-Network architecture.

As with DeepNEAT, because evaluation and training was limited to 8 epochs during evolution, the best network discovered by CoDeepNEAT must be able maximize performance within those limited number of epochs. While the network of Snoek et al. takes over 30 epochs to reach 20% test error and over 200 epochs to converge, the best network from evolution takes only 12 epochs to reach 20% test error and around 120 epochs to converge. This network utilizes the same modules multiple times, resulting in a deep and repetitive structure typical of many successful DNNs. A visualization of the

best evolved network is shown in Figure 4.7.

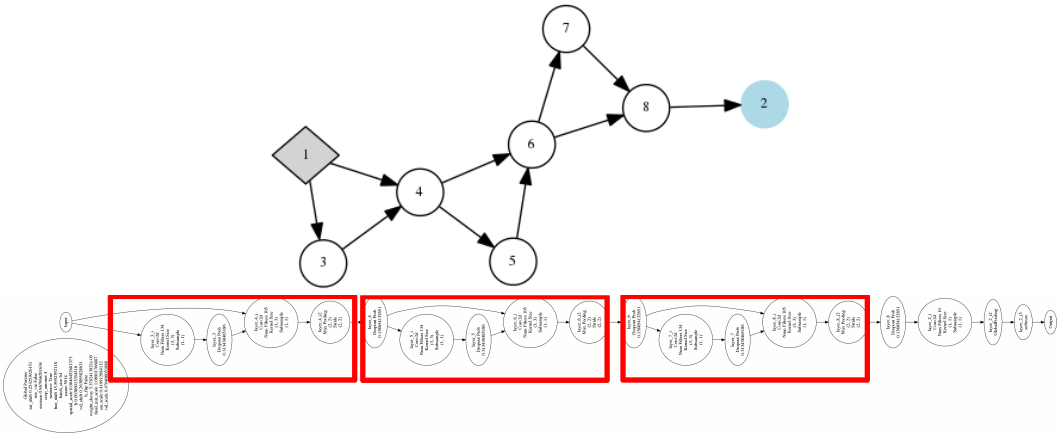


Figure 4.7: Top: High level visualization of the best network evolved by CoDeepNEAT for the CIFAR-10 domain. Node 1 is the input layer, while Node 2 is the output layer. The network has repetitive structure because its blueprint reuses same module in multiple places. Bottom: A more detailed visualization of the entire network with the locations of the modules highlighted in red. The use of modules allowed CoDeepNEAT to beat both DeepNEAT and a hand-designed architecture.

4.5 Experimental Results in MSCOCO Image Captioning Domain

This section with give an overview of the MSCOCO image captioning domain and report experimental results for both CoDeepNEAT and CoDeepNEAT-AES. In addition, a real world study is presented where a network evolved using CoDeepNEAT is applied to caption images online.

4.5.1 MSCOCO Domain Overview

Figure 4.8: Some examples of the types of images in the MSCOCO dataset: (a) iconic object images, (b) iconic scene images, and (c) non-iconic images [22].

Deep learning has recently provided state-of-the-art performance in image captioning, and several diverse architectures have been suggested [150,158, 164]. The input to an image captioning system is a raw image, and the output is a text caption intended to describe the contents of the image. In deep learning approaches, a convolutional network is usually used to process the image, and recurrent units, often LSTMs, to generate coherent sentences with long-range dependencies. Unlike image classification, the sequential nature of generating captions means that image captioning is a much harder task. It requires a more complicated network topology with many more different layer types and hyperparameters. As such, image captioning is a suitable second task to evaluate the effectiveness of CoDeepNEAT in discovering complex, diverse neural architectures often seen in the cutting edge of deep learning research.

The dataset used to evaluate the effectiveness of the candidate networks MSCOCO (Common Objects in Context) [22], a widely used benchmark dataset in image captioning. The dataset contains 500,000 captions for roughly 100,000 images that are generated by humans using Amazon Mechanical Turk. As seen in Figure 4.8, the images range from straightforward depictions of common objects in everyday to more complicated scenes where humans or animals interact with these objects.

4.5.2 Setup for MSCOCO Domain

Global Hyperparameter	**Range**
Learning Rate	$[0.0001, 0.1]$
Momentum	$[0.68, 0.99]$
Shared Embedding Size	$[128, 512]$
Embedding Dropout	$[0, 0.7]$
LSTM Recurrent Dropout	{True, False}
Nesterov Momentum	{True, False}
Weight Initialization	{Glorot normal, He normal}
Node Hyperparameter	**Range**
Layer Type	{Dense, LSTM}
Merge Method	{Sum, Concat}
Layer Size	$\{128, 256\}$
Layer Activation	{ReLU, Linear}
Layer Dropout	$[0, 0.7]$

Table 4.2: Node and global hyperparameters evolved for the image captioning case study. They show the size of the search space explored by CoDeepNEAT.

As is common in existing approaches, the evolved system uses a pre-trained ImageNet model to produce the initial image embeddings. The evolved network takes an image embedding as input, along with a one-hot text input.

During training, the text input contains the previous word of the ground-truth caption while during inference, it contains the previous word generated by the model.

In the initial CoDeepNEAT population the image and text inputs are fed to a shared embedding layer, which is densely connected to a softmax output over words. From this simple starting point, CoDeepNEAT evolves architectures that include fully connected layers, LSTM layers, sum layers, concatenation layers, and hyperparameters associated with each layer such as its size and dropout probabilities (Table 4.2). A set of global hyperparameters are also evolved, namely, the learning rate, momentum, and weight parameter initialization scheme. In order to have a baseline network that can be compared against, the search space for CoDeepNEAT was designed to include the state-of-the-art Show and Tell [150] image captioning architecture. The results in Section 4.5.3 show that CoDeepNEAT can beat the hand-designed architecture while using the same types of layers.

Since there is no single best accepted metric for evaluating captions, the fitness function is the mean across three commonly used metrics for evaluating caption quality (BLEU, METEOR, and CIDEr; [22]) normalized by their baseline values. The fitness is computed over a holdout set of 5000 images, i.e. 25,000 image-caption pairs.

These components and the glue that connects them were evolved as described in Section 4.2, with 100 networks trained in each generation. As with the image classification experiments, fitness evaluation and training of the net-

works were parallelized over 100 AWS EC2 workers equipped with GPUs. To keep the computational cost reasonable, during evolution the networks were trained for only six epochs, and only with a random 100,000 image subset of the 500,000 MSCOCO image-caption pairs. As a result, there was evolutionary pressure towards networks that converge quickly: The best evolved architectures trained to near convergence six times faster than the baseline Show and Tell model [150]. After evolution, the optimized learning rate was scaled by one-fifth to compensate for the subsampling. In addition, beam search of size three was used when generating captions for images in the test set.

4.5.3 Results for MSCOCO Domain for CoDeepNEAT

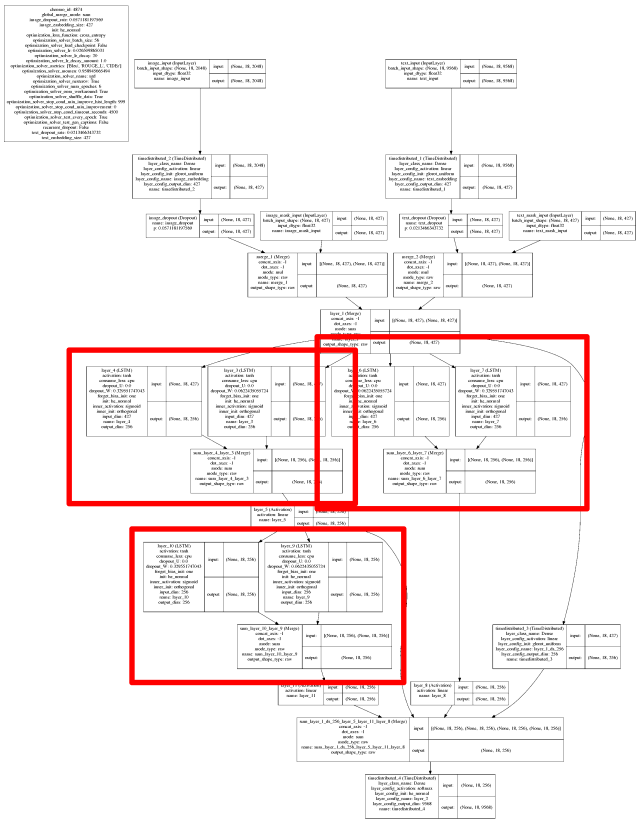


Figure 4.9: Visualization of the best architecture found by evolution. Among the components in its unique structure are six LSTM layers, four summing merge layers, and several skip connections. A single module architecture (highlighted in red) consisting of two LSTM layers merged by a sum is repeated three times. There is a path from the input through dense layers to the output that bypasses all LSTM layers, providing the softmax with a more direct view of the current input. The power of the architecture seems to come from the many shortcut connections, which are unlikely to have been discovered by hand.

Model	BLEU-4	CIDEr	METEOR
Neural Show and Tell [150]	27.7	85.5	23.7
DNGO* [137]	26.7	—	—
CoDeepNEAT*	**29.1**	**88.0**	**23.8**

Table 4.3: Summary of performance of different approaches on MSCOCO domain. The networks produced through architecture or hyperparameter search are labeled with an asterisk. The evolved network using CoDeepNEAT (highlighted in bold) improves over the hand-designed baselines and other architecture search methods. In particular, it was able to beat the Show and Tell network network by 5%.

Trained in parallel on about 100 GPUs, each generation took around one hour to complete. The most fit architecture was discovered on generation 37 (Figure 4.9). This architecture performed better than the hand-tuned baseline [150] when trained on the MSCOCO data alone (Table 4.3). In particular, the network evolved by CoDeepNEAT is able to beat Neural Show and Tell [150] by 5%, a previous state-of-the-art architecture with the same layer types and resides within the same search space that CoDeepNEAT explored.

Furthermore, the amount of time required to train the best evolved network was significantly less than the hand-designed baseline. While the baseline required roughly 60 epochs for the validation loss to converge, it only took about 10 epochs for the evolved network to do the same. The fast convergence speed of the evolved network was probably due CoDeepNEAT's bias towards evolving shortcut connections. The bias is due to the limited number of epochs of training during evolution; thus high performing networks had to evolve structure that maximized learning during those few epochs. The

skip connections often ends with a sum-merge layer, which shows similarity to residual architectures that are currently popular in deep learning [51, 145]. The many parallel, branching, and merging paths in the evolved network would have been very difficult to discover by hand.

4.5.4 Results on MSCOCO domain for CoDeepNEAT-AES

For comparison, two separate runs of CoDeepNEAT and CoDeepNEAT-AES were done on the MSCOCO image captioning domain. Except for parameters specific to CoDeepNEAT-AES, all other factors such as evolution configuration, search space, and network training/evaluation methods are kept identical. A population size of 100 was used for the synchronous version of CoDeepNEAT. For CoDeepNEAT-AES, $K = 300$ and $M = 100$ ($D = 3$) were used. The worker nodes were composed of up to 200 Amazon EC2 instances (with GPU support for training DNNs) and the completion service provides the interface between them and the server. Due to cost concerns of running so many EC2 instances, a smaller value of $D = 3$ was used. Because EC2 spot instances are inherently unreliable and may be temporary unavailable for any reason, both experiments were started at the same time to remove a potential source of bias.

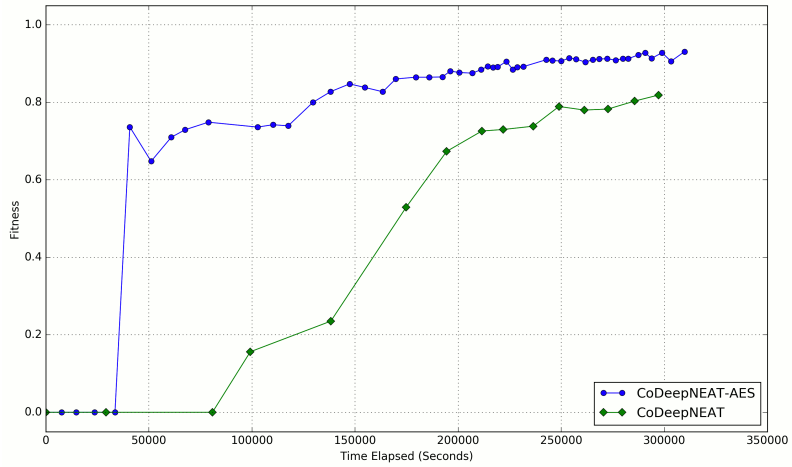


Figure 4.10: A plot of fitness versus time elapsed for synchronous CoDeepNEAT and CoDeepNEAT-AES. Each marker in the plot represents the fitness at a different generation. At any given time, CoDeepNEAT-AES was able to achieve much better fitness.

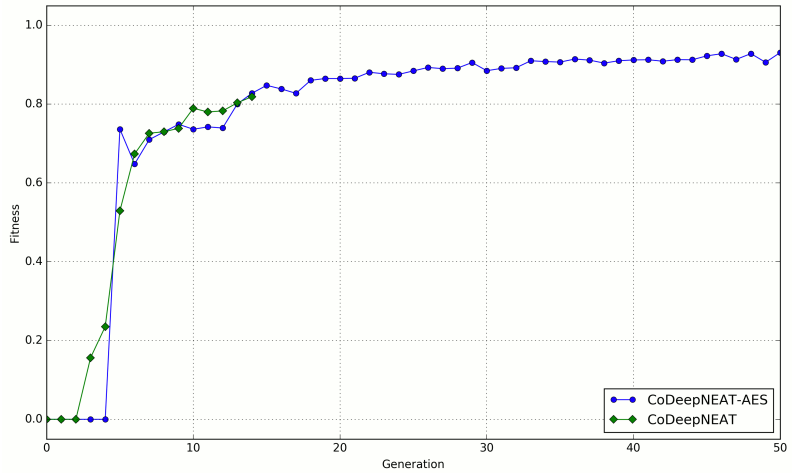


Figure 4.11: A plot of fitness versus number of generations elapsed for synchronous CoDeepNEAT and CoDeepNEAT-AES. This result shows that both algorithms achieved the same fitness when compared by generations. However, the generations for CoDeepNEAT-AES were much shorter in duration.

Figure 4.10 shows that when both CoDeepNEAT and CoDeepNEAT-AES are given the same amount of time, CoDeepNEAT-AES is able to reach better fitness and converge faster. Although both versions of CoDeepNEAT achieve similar fitness after the same number of generations (Figure 4.11), each generation of synchronous CoDeepNEAT takes far longer; this can be seen in the plot of fitness versus the amount of time elapsed. Due to time constraints, only CoDeepNEAT-AES was allowed to run to convergence. However, the converged fitness value for CoDeepNEAT-AES (0.93) is essentially identical to the converged fitness of a previous run of synchronous CoDeepNEAT. Overall, the experiment results suggest that CoDeepNEAT-AES can accelerate the performance of CoDeepNEAT by up to three times in the image captioning domain.

4.6 Experimental Results in Wikidetox Comment Classification Domain

This section gives an overview of the Wikidetox comment classification problem. CoDeepNEAT was used to evolve a network in this domain and state-of-the-art experimental results are reported. [1]

4.6.1 Wikidetox Domain Overview

Wikipedia is one of the largest encyclopedias that is publicly available online [11], with over 5 million written articles for the English language alone.

[1] The Wikidetox experiments were ran with the assistance of Elliot Meyerson.

Unlike traditional encyclopedias, Wikipedia can be edited by any user who registers an account. It is common for controversial or sensitive articles to cause disagreements between various editors. As a result, in the discussion section for some articles, there are often vitriolic or hateful comments that are directed at other users. These comments are commonly referred to as "toxic" and it has become increasing important to detect toxic comments and remove them from Wikipedia.

The Wikipedia Detox dataset (Wikidetox) is a collection of 160K example comments that are divided into 93K train, 31K validation, and 31K test examples [6]. The labels for the comments are generated by humans using crowd-sourcing methods and contain four different categories for toxic comments. However, following previous work from Chu at el. [23], all toxic comment categories are combined together, thus creating a binary classification problem. The dataset is also unbalanced with only about 9.6% of the comments actually being labeled as toxic.

4.6.2 Setup for the Wikidetox Domain

The experimental setup for the Wikidetox domain was similar to the MSCOCO image captioning domain. The layer types and hyperparameters that CoDeepNEAT was allowed to evolve was the same as those listed for the MSCOCO domain in Table 4.2. Like in the MSCOCO domain, the search space for network work architectures was defined around recurrent (LSTM) layers as the basic building block. Since comments are essentially an ordered

list of words, recurrent layers (having shown to be effective at processing sequential data) were a natural choice. In order for the words to be given as input into a recurrent network, they must be converted into an appropriate vector representation first. Before given as input to the network, the comments were preprocessed using a recently introduced method for generating word embeddings called FastText [18], which itself is an improvement upon the more commonly used Word2Vec [101].

During evolution, the blueprint and module population sizes for CoDeepNEAT were 25 and 50 respectively, with 100 assembled networks generated at every generation. For evaluation, each evolved DNN was trained for three epochs and the classification accuracy was returned back as the fitness. Preliminary experiments showed that three epochs were enough for the network to converge during training. Thus, unlike the CIFAR-10 and MSCOCO domains, there was no need for an extra step where the best evolved network is trained fully to convergence. Like in the previous experiments, the evaluation of DNNs at every generation was distributed over 100 worker machines.

4.6.3 Experimental Results in the Wikidetox Domain

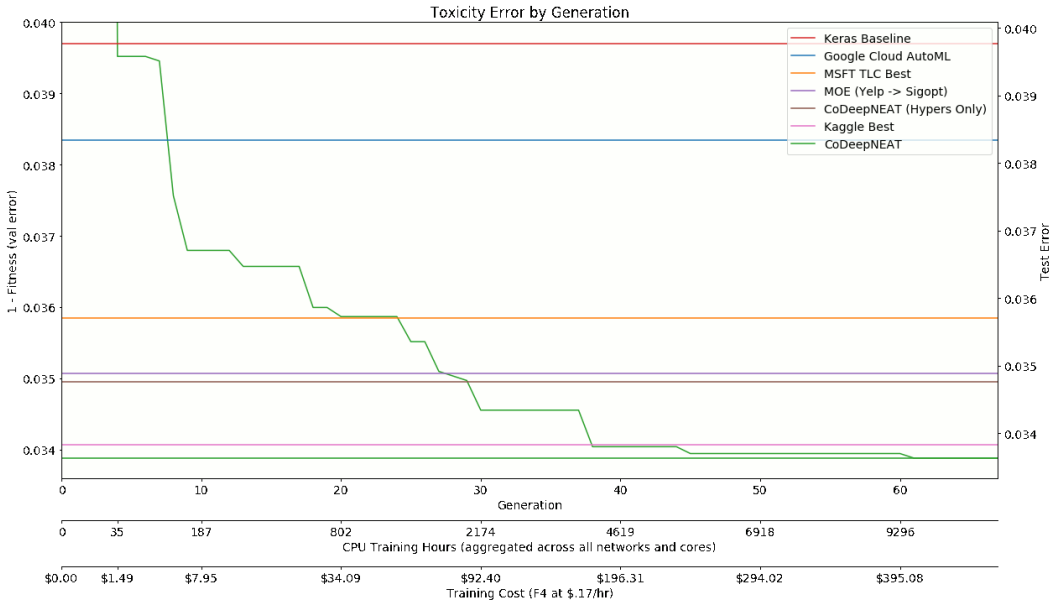


Figure 4.12: A comparison of CoDeepNEAT against the networks discovered via several commercially available methods, including Kaggle, MSFT TLC, MOE, and Google AutoML. The Y-axis shows best fitness/accuracy achieved so far, while the X-axis shows the generations, total training time, and total amount of money spent on cloud compute. As the plot shows, CoDeepNEAT is gradually able to discover better networks, eventually finding one in the 40th generation that beats all other approaches.

Wikidetox is an interesting domain since it was part of a Kaggle challenge; as a result, there already exists several hand-designed networks to compare against [2]. Furthermore, due to the relative speed at which networks can be trained on this dataset, it was practical to evaluate hyperparameter optimization methods from companies such as Microsoft and Google on the dataset. As a result, the networks evolved by CoDeepNEAT can be com-

pared against networks designed by the best commercially available automatic machine learning frameworks.

Figure 4.12 shows a comparison of CoDeepNEAT against the following approaches: (1) A simple baseline network from the Kaggle competition, (2) the best hand-designed architecture from the Kaggle competition, (3) Microsoft's TLC library for hyperparameter optimization [10], (4) MOE, a Bayesian optimization algorithm from Yelp (commercialized by SigOpt) [4], (5) Google AutoML [12], and (6) a stripped down version of CoDeepNEAT which only optimizes hyperparameters and not the architecture. The figure shows how the performance of best network discovered by CoDeepNEAT improved over the generations and also the amount of training time and cost spent so far.

Initially, the best network architectures discovered by CoDeepNEAT were only barely better than the simple Kaggle baseline network. However, CoDeepNEAT was eventually able to outperform all the other approaches. By generation 40, the best evolved network was able to beat the Kaggle competition network, which is the current state-of-the-art on the Wikidetox dataset. What is interesting about CoDeepNEAT is that there are clear trade-offs between the amount of training time/money used and the quality of the results. Depending on the budget available, an user running CoDeepNEAT can stop earlier to get results competitive with existing approaches such as TLC or AutoML or run CoDeepNEAT to convergence to get the best possible results. Moreover, the results demonstrate the value of architecture optimization

over just hyperparameter optimization. While MOE and hyperparameter-only CoDeepNEAT perform similarly, full CoDeepNEAT performs significantly better.

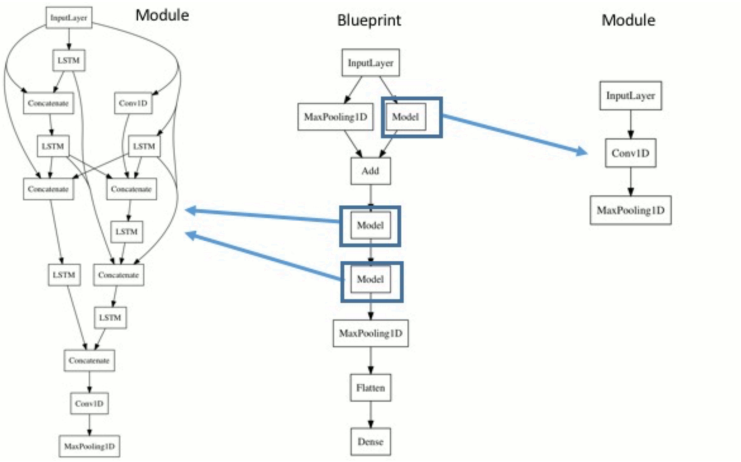


Figure 4.13: A visualization of the best network discovered by CoDeepNEAT in the Wikidetox domain. As the network architecture shows, it is most optimal to use a combination of both simple and complex modules within the blueprint of the network.

A visualization of best performing network discovered by CoDeepNEAT in the Wikidetox domain is shown in Figure 4.13. This network architecture is composed of a relatively linear blueprint that uses one simple module and two instances of a complicated module architecture. The complicated module contains many shortcut connections and branching parallel paths between the layers. As with the MSCOCO and CIFAR-10 domains, it appears that these shortcut connections and branching paths accelerate learning and allow for faster convergence.

4.7 Real-World Use of DNNs Evolved with CoDeepNEAT

One important aspect of CoDeepNEAT that has not been explored is its effectiveness in building real-world deep learning applications. This section will explore using CoDeepNEAT to evolve a image captioning network for helping blind people recognize images that are often shown in popular online websites.

Unlike most real world images, the MSCOCO dataset is chosen to depict "common objects in context". The focus is on a relatively small set of objects and their interactions in a relatively small set of settings. The internet as a whole, and an online magazine website in particular, contain many images that cannot be classified as "common objects in context". Other types of images from the magazine include staged portraits of people, infographics, cartoons, abstract designs, and *iconic* images, i.e. images of one or multiple objects *out of context* such as on a white or patterned background. Therefore, an additional dataset of 17,000 image-caption pairs was constructed for the case study, targeting iconic images in particular. First, 400 images were first scraped from Wired.uk, a popular technology magazine website, and 1000 of them were identified as iconic. Then, 16,000 images that were visually similar to those 1000 were retrieved automatically from a large image repository. A single ground-truth caption for each of these 17K images was generated by human subjects from MightyAI [3]. The holdout set for evaluation consisted of 100 of the original 1000 iconic images, along with 3000 other images.

During evolution, networks were trained and evaluated only on the MSCOCO data. The best architecture from evolution was then trained from scratch on both the MSCOCO and MightyAI datasets in an iterative alternating approach: one epoch on MSCOCO, followed by five epochs on MightyAI, until maximum performance was reached on the MightyAI holdout data. Beam search was then used to generate captions from the fully trained models. Performance achieved using the MightyAI data demonstrates the ability of evolved architectures to generalize to domains towards which they are not evolved.

Once the model was fully-trained, it was placed on a server where it could be queried with images to caption. A JavaScript snippet was written that a developer can embed in his/her site to automatically query the model to caption all images on a page [2]. This snippet runs in an existing Chrome extension for custom scripts and automatically captions images as the user browses the web. These tools add captions to the 'alt' field of images, which screen readers can then read to blind internet users (Figure 4.14).

[2]Thanks to Olivier Francon for the script

Figure 4.14: An iconic image from an online magazine captioned by an evolved model. The model provides a suitably detailed description without any unnecessary context.

However, a more important result is the performance of this network on the magazine website. Because no suitable automatic metrics exist for the types of captions collected for the magazine website (and existing metrics are very noisy when there is only one reference caption), captions generated by the evolved model on all 3100 holdout images were manually evaluated on a scale from one to four (Figure 4.15). The model performed reasonably on iconic images, where it accurately captioned roughly half of them. Unfortunately it did not too well on more general images and was only able to correctly caption roughly 20% of such images.

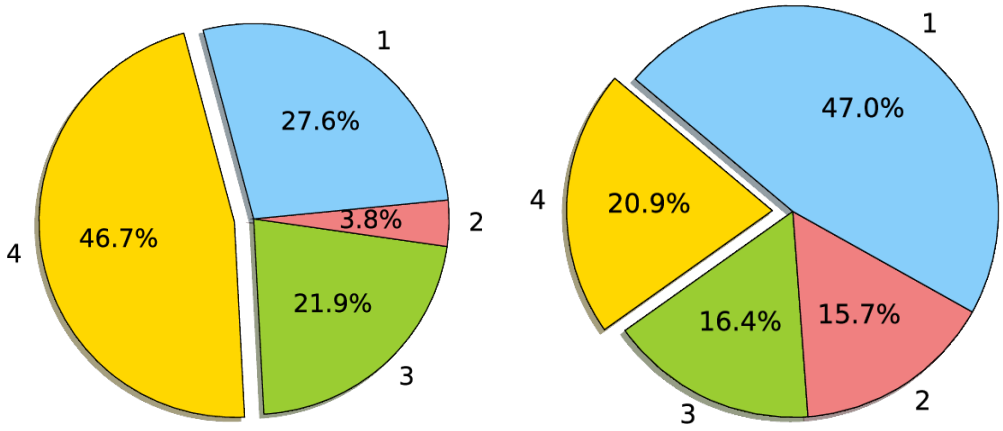
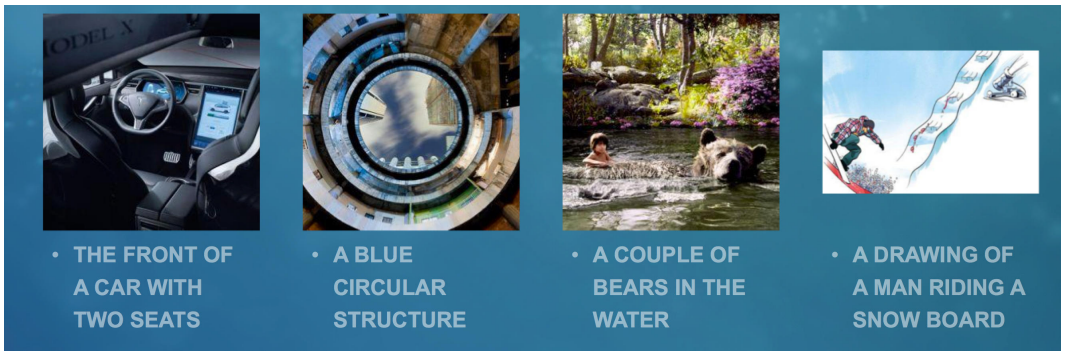
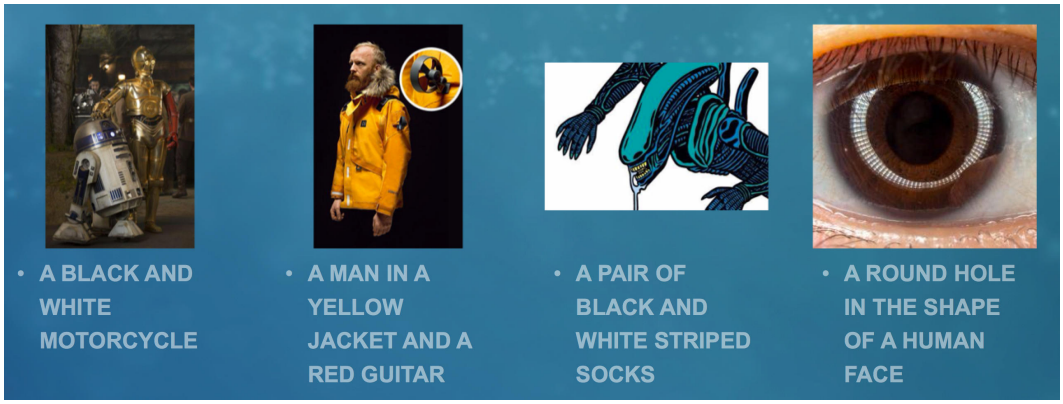


Figure 4.15: Results for captions generated by an evolved model for the online magazine images rated from 1 to 4, with 4=Correct, 3=Mostly Correct, 2=Mostly Incorrect, 1=Incorrect. Left: On iconic images, the model is able to get about one half correct; Right: On all images, the model gets about one fifth correct. The superior performance on iconic images shows that it is useful to build supplementary training sets for specific image types.

Figure 4.16 shows some examples of good and bad captions for these images. The images that were accurately captioned typically are simpler in content, contain few objects in the foreground of the image, and share similarities with images present in the MSCOCO dataset. On the other hand, the images which could not be captioned properly are more abstract or contained objects typically not seen in the MSCOCO dataset. Nevertheless, the bad captions are not always completely wrong and there often are some details in the image which are preserved in the caption. One way to boost performance in the future might be to add more such images into the training set.



(a) Good captions



(b) Bad captions

Figure 4.16: Top: Four good captions. The model is able to abstract about ambiguous images and even describe drawings, along with photos of objects in context. Bottom: Four bad captions. When it fails, the output of the model still contains some correct sense of the image. The results overall are promising and suggest that the model can be improved by including more difficult images in the training set.

The model is not perfect, but the results are promising. There are many known improvements that can be implemented, including ensembling diverse architectures generated by evolution, fine-tuning of the ImageNet model, us-

ing a more recent ImageNet model, and performing beam search or scheduled sampling during training [151]. For this application, it is also important to include methods for automatically evaluation caption quality and filtering captions that would give an incorrect impression to a blind user. However, even without these additions, the results demonstrate that it is now possible to develop practical applications through evolving DNNs.

4.8 Conclusion

This chapter introduced CoDeepNEAT, a coevolutionary EA for efficiently evolving DNNs, and CoDeepNEAT-AES, the asynchronous version of CoDeepNEAT. Experimental results in both the CIFAR-10 and MSCOCO domains validated that CoDeepNEAT can evolve networks that outperform previous state-of-the-art hand-designed architectures and even those evolved with DeepNEAT. The best discovered networks not only have good performance, but are also quick to train as well. By using CoDeepNEAT to optimize a network to caption images found on popular websites, it shows that evolutionary architecture search is useful for building real-world applications.

Chapter 5

Coevolution of Deep Neural Network Architectures for Multitask Learning

This chapter is organized as follows: (1) First, the motivation behind modifying CoDeepNEAT to the multitask learning (MTL) domain is explored. (2) Afterwards, soft-ordering and how it can be applied to MTL is reviewed. (3) Next, CodeepNEAT is extended and modified to incorporate elements from the Soft-ordering architecture to evolve better multitask architectures. (4) This improved version, which has been adapted to MTL, is then applied to two popular MTL domains, namely the Omniglot character recognition dataset and the medical Chest X-ray image classification dataset. [1]

5.1 Motivation

So far, CoDeepNeat has been applied to evolve and optimize the architecture of deep neural networks (DNN) on a single task. In this chapter, CoDeepNEAT is modified to search for multitask neural network architectures. It is already possible to apply CoDeepNEAT to the multitask domain

[1] Some work in this chapter was previously published [84]. The author's specific contributions included the CM and CMSR algorithms.

by simply forcing the last layer of the evolved network to have a multi-head decoder, one for each task. This is the architecture commonly used in classical MTL architectures [19] and often copied in many DNN applications of MTL. However, this architecture would force all the tasks to share the same weights except for the last layer and as shown by Meyerson, et al. [99], such a design is not optimal and can be improved.

This chapter will describe how CoDeepNEAT can be leveraged to evolve non-trivial multitask architectures that perform significantly better in a variety of MTL domains. One straightforward way is to generalize the soft-ordering architecture described in [99] by allowing CoDeepNEAT to evolve how the different components are routed and connected with each other. Figure 5.1 provides an overview of how this generalization is accomplished. The foundation is (1) the original soft-ordering, which uses a fixed architecture for the modules and a fixed routing (i.e. network topology) that is shared among all tasks. This architecture is then extended in two ways with CoDeepNEAT: (2) by coevolving the module architectures (CM), and (3) by coevolving both the module architectures and a single shared routing for all tasks using (CMSR). Figure 5.3 gives high-level algorithmic descriptions of these methods, which are described in more detail in Sections 5.3 and 5.4.

It is interesting to note that there is another algorithm similar to CMSR called Coevolution of Task Routings (CTR) [84]. This algorithm evolves separate routings for each task and has been combined with CM to produce state-of-the-art results on the Omniglot domain.

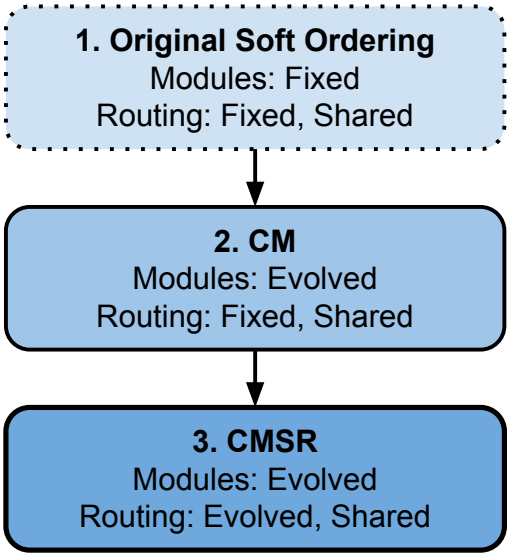


Figure 5.1: The relationships of various MTL architectures described in this chapter. The soft-ordering method [99] is used as the starting point, extending it with CoDeepNEAT, leading to CM and CMSR on the bottom.

5.2 Soft-ordering Architecture

One way to improve upon the classical MTL DNN architecture is to apply soft-ordering of layers to DNN architectures [99]. This approach allows shared layers to be used across different depths in an unique manner for different tasks. Through backpropagation, the joint model learns how to use each shared (potentially nonlinear) layer W_d at each depth d for the t-th task. This idea is implemented by learning a distinct scalar s_{tdl} for each such location, which then multiplies the layer's output. The final output at depth d for the task is then the sum of these weighted outputs across layers, i.e., a *soft-merge*.

More generally, a soft-merge is a learnable function given by

$$\text{softmerge}(\text{in}_1, \ldots, \text{in}_M) = \sum_{m=1..M} s_m \text{in}_m, \text{ with } \sum_{m=1..M} s_m = 1\,, \tag{5.1}$$

where the in_m are a list of incoming tensors, s_m are scalars trained simultaneously with internal layer weights via backpropagation, and the constraint that all s_m sum to 1 is enforced via a softmax function. Figure 5.2 shows an example soft-ordering network.

More formally, given shared layers $W_1, W_2, \ldots, W_D$, the soft-ordering model $\boldsymbol{y}_t = f(\boldsymbol{x}_t)$ for the t-th task $\{(\boldsymbol{y}_{ti}, \boldsymbol{x}_{ti})\}_{i=1}^N$ is given by $\boldsymbol{y}_t = \mathcal{D}_t(\boldsymbol{y}_t^D)$, where $\boldsymbol{y}_t^0 = \mathcal{E}_t(\boldsymbol{x}_t)$ and

$$\boldsymbol{y}_t^d = \text{softmerge}\big(W_1(\boldsymbol{y}_t^{d-1}), \ldots, W_D(\boldsymbol{y}_t^{d-1})\big) \quad \forall\, d \in 1..D\,, \tag{5.2}$$

where $\mathcal{E}_t$ is a task-specific encoder mapping the task input to the input of the shared layers, $\mathcal{D}_t$ is a task-specific decoder mapping the output of the shared layers to an output layer, e.g., classification.

Although soft-ordering allows flexible sharing across depths, layers are still only applied in a fixed grid-like topology, which biases and restricts the type of sharing that can be learned. A visual overview of soft-ordering is given below in Figure 5.2. This chapter will use CoDeepNEAT to generalize soft-ordering layers to more general modules, and introduces evolutionary approaches to both design these modules and to discover how to assemble these modules into appropriate topologies for multitask learning.

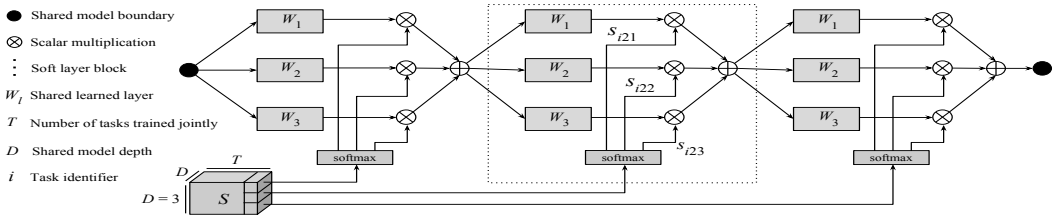


Figure 5.2: Example soft-ordering network with three shared layers [99]. Soft-ordering learns how to use the same layers in different locations by learning a tensor S of task-specific scaling parameters. S is learned jointly with the W_d, to allow flexible sharing across tasks and depths. This architecture enables the learning of layers that are used in different ways at different depths for different tasks.

5.3 Coevolution of Modules

In Coevolution of Modules (CM), CoDeepNEAT is used to search for promising module architectures, which then are inserted into appropriate positions to create an *enhanced soft-ordering* network.

5.3.1 Algorithm Overview

The evolutionary process for CM is described in detail below:

1. CoDeepNEAT initializes a population of modules P_M. The blueprints are not used.

2. Modules are randomly chosen from each species in P_M, grouped into sets M and are assembled into enhanced soft-ordering networks.

3. Each assembled network is trained/evaluated on some task and its performance is returned as fitness.

Algorithm 5 CM (Section 5.3)

1. **Given** fixed blueprint
2. **Initialize** module population
3. **Each** generation:
   4. **Assemble** MTL networks with modules
   5. **Randomly initialize all weights**
   6. **Train** each MTL network with backprop
   7. **Assign fitnesses** to modules
   8. **Update** module populations

Algorithm 6 CMSR (Sec. 5.4)

1. **Initialize** blueprint/module populations
2. **Each** generation:
   3. **Assemble** MTL networks with blueprints/modules
   4. **Randomly initialize all weights**
   5. **Train** each MTL network with backprop
   6. **Assign fitnesses** to modules and blueprints
   7. **Update** blueprint/module populations

Figure 5.3: High-level algorithm outlines of CM and CMSR, illustrating how they are similar and different from each other. In particular, the algorithms differ in whether or not the blueprint is evolved along with the modules (see line 7 in Algorithm 6).

4. Fitness is attributed to the modules, and NEAT evolutionary operators are applied to evolve the modules.

5. The proess is repeated from Step 1 until CoDeepNEAT terminates, i.e. no further progress is observed for a given number of generations.

Unlike in soft-ordering [99], the number of modules and the depth of the network are not fixed but are evolved as global hyperparameters by CoDeepNEAT (however the layout is still a grid-like structure). Since the routing layout is fixed, the blueprint population of CoDeepNEAT, which determines how the modules are connected, is not used. Thus one key operation in the original CoDeepNEAT, i.e. inserting modules into each node of the blueprint DAG, is skipped; only the module population is evolved.

To assemble a network for fitness evaluation, an individual is randomly chosen from each species in the module population to form an ordered set of distinct modules M. These modules are NEAT graphs where each node represents a particular convolutional layer and associated hyperparameters. The hyperparameters evolved in each of the module's layers include the activation function, kernel size, number of filters, L2 regularization strength and output dropout rate. In addition, CoDeepNEAT also coevolves global hyperparameters that are relevant to the entire assembled network as a whole; these include learning rate, the number of filters of the final layer of each module, and the weight initialization method. Evolvable hyperparameters in each node include the activation function, kernel size, number of filters, L2 regulariza-

tion strength and output dropout rate. The modules are then transformed into actual neural networks by replacing each node in the DAG with the corresponding layer. To ensure compatibility between the inputs and outputs of each module, a linear 1×1 convolutional layer (number of filters determined by a global hyperparameter), followed by a max-pooling layer (provided that the feature map before pooling is at least 4×4) is included as the last layer in each module.

The modules are then inserted into the soft-ordering network. The architecture of the network is interpreted as a grid of $K \times D$ slots, where d indicates the depth of the network and the slots with the same k value have the same module topology. For each available slot T_{kd}, the corresponding module M_k is inserted. If $k > |M|$, then $M_{k \bmod |M|}$ is inserted instead.

5.3.2 Weight Sharing Between Modules

Each module in a particular slot has the potential to share its weights with modules that have the same architecture and are located in other slots of the blueprint. Flag F_k in each module indicates whether or not the module's weights are shared. This flag is evolved as part of the module genotype in CoDeepNEAT. Also, there is also global flag F_d for each depth of the soft-ordering network. If the M_k is placed in T_{kd} and both F_k and F_d are turned on, then the module is able to share its weights with any other M_k whose slot have both flags turned on as well. Such an arrangement allows each slot to have sharing enabled and disabled independently.

The assembled network is attached to separate encoders and decoders for each task and trained jointly using a gradient-based optimizer. Average performance over all tasks is returned as fitness back to CoDeepNEAT. That fitness is assigned to each of the modules in the assembled network. If a module is used in multiple assembled networks, their fitnesses are averaged into module fitness. After evaluation is complete, standard NEAT mutation, crossover, and speciation operators are applied to create the next generation of the module population [141].

5.4 Coevolution of Modules and Shared Routing

Coevolution of Modules and Shared Routing (CMSR) extends CM to include blueprint evolution.

5.4.1 Algorithm Overview

IN CMSR, the routing between various modules no longer follows the fixed grid-like structure, but instead an arbitrary DAG. Each node in the blueprint genotype points to a particular module species. During assembly, the blueprints are converted into deep multitask networks as follows:

1. For each blueprint in the population, an individual module is randomly chosen from each species.

2. Each node in the blueprint is then replaced by the module from the appropriate species.

3. If a module has multiple inputs from previous nodes in the blueprint, the inputs are soft-merged first [99].

4. The process is repeated from step 1 until reaching a target number of assembled networks.

As in CM, each node in the blueprint has a flag F_i that indicates whether node N_i should be shared or not. If two nodes are replaced by the same module and if both nodes have the sharing flag turned on, then the two modules will share weights. Such an arrangement allows each node to evolve independently whether to share weights or not. The training procedures for both CM and CMSR are otherwise identical. After fitness evaluation, the fitness is assigned to both blueprints and modules in the same manner as with CM. To accelerate evolution, the blueprint population is not initialized from minimally connected networks like the modules, but from randomly mutated networks that on average have five nodes.

5.4.2 Random Population Initialization

One main concern regarding CMSR compared to CM is that the search space has been massively increased. In CM, only module architectures are searched and evolved, but in CMSR, both the module architectures and shared routing between the modules have to be optimized. As a result, if both CM and CMSR are to start evolving from minimal topologies, CMSR takes significantly longer than CM to converge. One way to accelerate search is to allow CMSR to start with populations of randomly initialized blueprint/module

genes of some arbitrary complexity. First, blueprint and module populations of minimal complexity are initialized. Then, the NEAT add-connection and add-node mutation operators are repeatedly applied with some probability P to each blueprint and module K times. Afterwards, evolution is then allowed to proceed as normal. Preliminary results show that this initialization method creates a diverse population with varying complexity and significantly closes the gap in convergence speed between CMSR and CM.

5.5 Experimental Results for Omniglot Multitask Learning Domain

This section will describe experimental setup and results for the Omniglot character recognition MTL domain.

5.5.1 Omniglot Domain Overview

The Omniglot dataset consists of 50 alphabets of handwritten characters [75], each of which induces its own character recognition task. There are 20 instances of each character, each a 105×105 black and white image. Omniglot is a good fit for MTL, because there is clear intuition that knowledge of several alphabets will make learning another one easier. Omniglot has been used in an array of settings: generative modeling [75, 122], one-shot learning [71, 75, 132], and deep MTL [17, 96, 99, 120, 161]. Previous deep MTL approaches used random training/testing splits for evaluation [17, 99, 161]. However, with model search (i.e. when the model architecture is learned as well), a validation set

separate from the training and testing sets is needed. Therefore, in the experiments in this paper, a fixed training/validation/testing split of 50%/20%/30% is introduced for each task. Because training is slow and increases linearly with the number of tasks, a subset of 20 tasks out of the 50 possible is used in the current experiments. These tasks are trained in a fixed random order. Soft-ordering is the current state-of-the-art method in this domain [99]. The experiments therefore used soft-ordering as a starting point for designing further improvements.

5.5.2 Setup for Omniglot Domain

For CoDeepNEAT fitness evaluations, all networks were trained using Adam [70] for 3000 iterations over the 20 alphabets. Each iteration is equivalent to one full forward and backward pass through the network with a single example image and label chosen randomly from each task. The fitness assigned to each network was the average validation accuracy across the 20 tasks after training. No data augmentation was performed during training during evolution.

For CM and CMSR, CoDeepNEAT was initialized with approximately 50 modules (in four species) and 20 blueprints (in one species). During each generation, 100 networks were assembled from modules and/or blueprints for evaluation. The global and layer-specific evolvable hyperparameters are described in Section 5.3. With CoDeepNEAT, the evaluation of assembled networks was distributed over 100 separate EC2 instances with a K80 GPU in

Algorithm	Val Accuracy (%)	Test Accuracy (%)
1. Single Task [99]	63.59 (0.53)	60.81 (0.50)
2. Soft-ordering [99]	67.67 (0.74)	66.59 (0.71)
3. CM	80.38 (0.36)	81.33 (0.27)
4. CMSR	83.69 (0.21)	83.82 (0.18)
5. CMSR (Data Aug)	**92.16 (0.16)**	**91.57 (0.15)**

Table 5.1: Average validation and test accuracy over 20 tasks for each algorithm. CMSR performs the best as it combines both module and routing evolution. Pairwise t-tests show all differences are statistically significant with $p < 0.05$.

AWS. The average time for training was usually around 1-2 hours depending on the network size.

Because the fitness returned for each assembled network was noisy, to find the best assembled CoDeepNEAT network, the top 50 highest fitness networks from the entire history of the run were retrained for 30,000 iterations. For the CM and CMSR experiments, decaying the learning rate by a factor of 10 after 10 and 20 epochs of training gave a moderate boost to performance. There was also an option to enable data augmentation, where the image was randomly shifted horizontally and vertically 16 pixels before given as input to the network. To evaluate the performance of the best assembled network on the test set (which is not seen during evolution or training), the network was trained from scratch again for 30,000 iterations. This snapshot was then evaluated and the average test accuracy over all tasks returned.

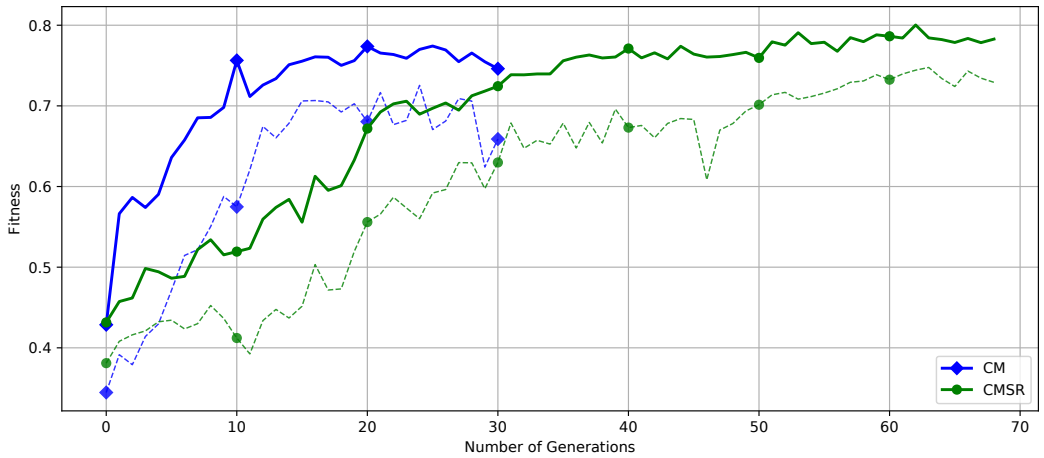


Figure 5.4: Comparison of fitness (validation accuracy after partial training for 3000 iterations) over generations of single runs of CM and CMSR. Solid lines show the fitness of best assembled network and dotted line show the mean fitness. Both methods reach a similar fitness, but CMSR is slower to converge.

5.5.3 Results for Omniglot Domain

Figure 5.4 demonstrates how the best and mean fitness improves for CM and CMSR in the CoDeepNEAT outer loop where module/blueprint co-evolution occurs. While networks evolved by CMSR were able to reach a higher accuracy when trained fully, both algorithms converged roughly to the same final fitness value during evolution, which is around 78% validation accuracy. However due to its smaller search space, CM converged faster than CMSR. This result is expected since the search space of CM (evolution of modules and weight sharing) is smaller than CMSR (evolution of modules, blueprints, and weight sharing).

One open question is how much sharing of weights between modules

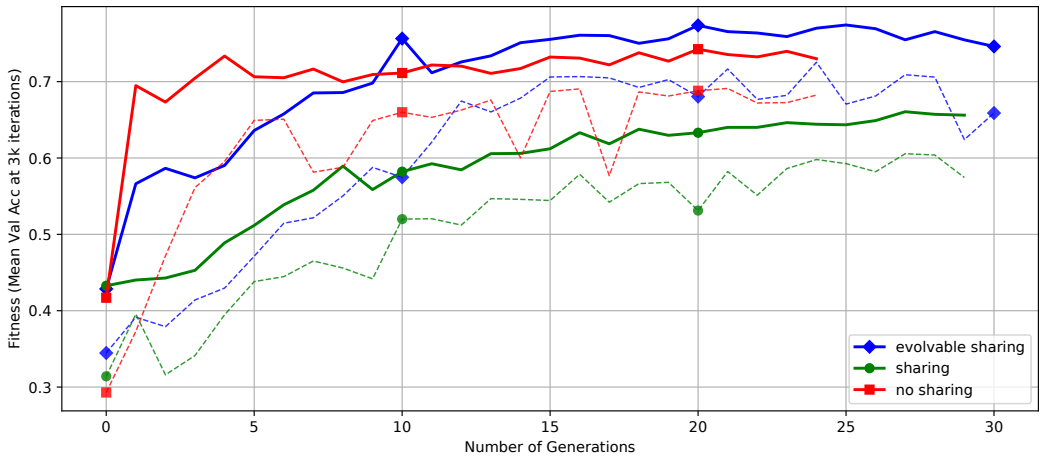


Figure 5.5: Comparison of fitness over generations of CM with disabling, enabling, and evolving module weight sharing. No sharing is better than forced sharing, but evolvable sharing outperforms them both, validating the approach.

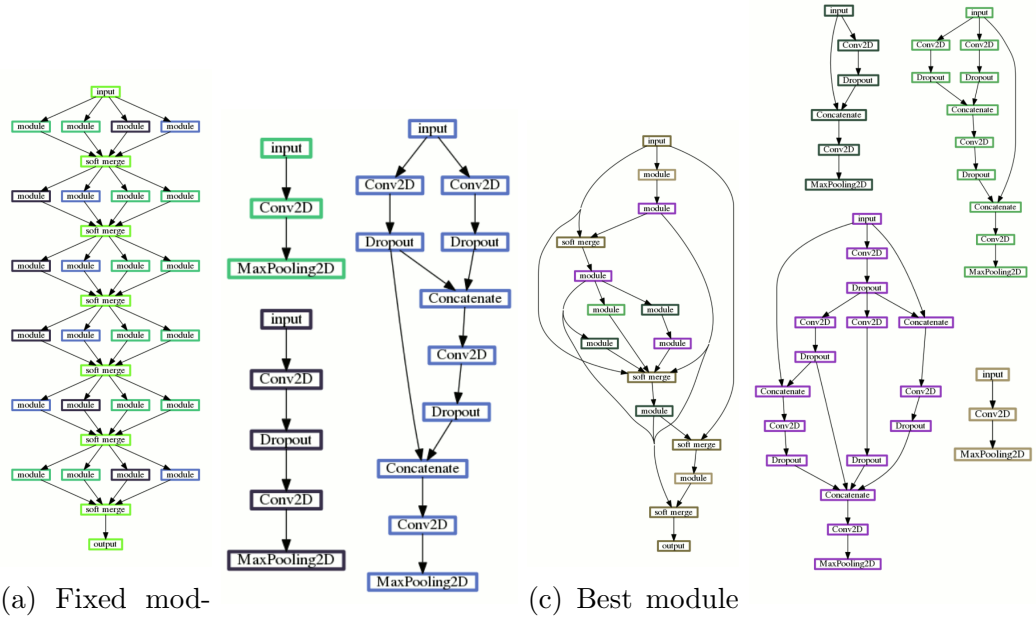
affects the performance of the assembled network. Figure 5.5 compares the effect of enabling, disabling, and evolving weight sharing with CM. Interestingly, disabling weight sharing led to better performance than enabling it, but evolving it is best. Thus, the design choice of evolving sharing in CM and CMSR is vindicated. An analysis of the architecture of the best assembled networks shows that weight sharing in particular locations such as near the output decoders is a good strategy.

Table 5.1 shows the validation and test accuracy for the best evolved network produced by each method, averaged over 10 runs. The best-performing methods are highlighted in bold and standard error for the 10 runs is shown in parenthesis. In addition, performance of the baseline methods are shown,

namely (1) a hand-designed single-task architecture, i.e. where each task is trained and evaluated separately, and (2) the soft-ordering network architecture [99]. Indeed, the methods improve upon the baseline according to increasing complexity: Evolving modules and evolving topologies is significantly better than the baselines, and evolving both is significantly better than either alone. Interestingly, data augmentation during training gave CMSR a massive boost in performance and allowed CMSR to overtake similar methods such as CMTR (combination of CTR and CM) [84] in the Omniglot domain.

Figure 5.6a and 5.6b displays the module routing and architectures of the best network evolved using CM on the Omniglot domain. In the visualization, each unique module architecture is associated with a particular color; this color is then used to show what architecture is used at different locations in the overall network. The routing between the modules is relatively fixed, only the depth and width of the arrangement of the modules can change. However, the modules themselves can have arbitrary structure and as seen by their visualizations, there is a mixture of both complex and simple module architectures.

Similarly, Figure 5.6c and 5.6d visualizes the best network that was evolved using CMSR. In this architecture, the routing between the module is fully evolvable and is characterized by lots of shortcut connections and branching paths. Similar to CM, the modules evolved using CMSR are also a mix of both complex and simple structure.



(a) Fixed module routing used by CM.

(b) Best modules evolved by CM.

(c) Best module routing evolved by CMSR.

(d) Best modules evolved by CMSR.

Figure 5.6: Visualizations of the best networks evolved by CM (Figures 5.6a and 5.6b) and CMSR (Figures 5.6c and 5.6d) on the Omniglot domain. The module routing for CM is fixed but is evolvable in CMSR. The module routing (blueprint) evolved by CMSR contains many shortcut connections and could be a possible factor in CMSR's superior performance. Both methods evolve a mixture of both complex and simple module architectures.

5.6 Experimental Results for Chest X-Ray Multitask Learning Domain

This section will describe experimental setup and results for the Chest X-Ray MTL domain.

5.6.1 Chest X-ray Domain Overview

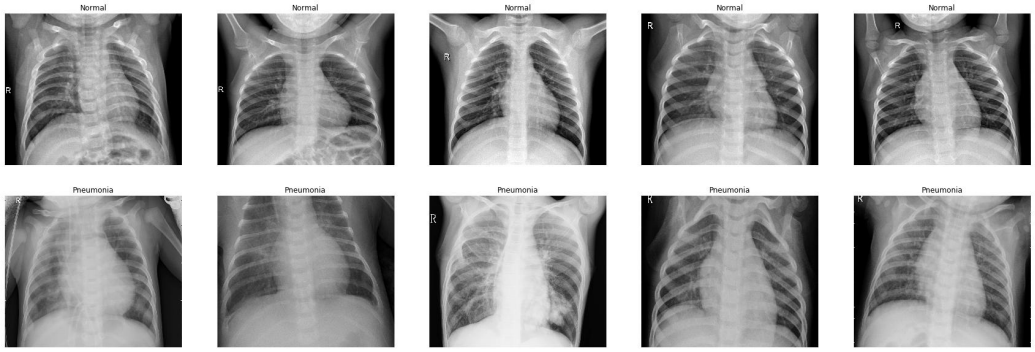


Figure 5.7: The top row shows negative examples of images from the Chest X-ray dataset where no disease is present. The bottom row show positive examples where the images do show a disease.

Chest X-ray is a recently introduced MTL benchmark that involves classifying x-ray images of the chest region of various patients [114, 152]. The dataset is composed of 112,120 high resolution frontal chest X-ray images, and the images are labeled with one or more of 14 different diseases, or no diseases at all. The multi-label nature of the dataset naturally lends to a MTL setup where each disease is an individual binary classification task. Both positive and negative example images from the Chest X-ray dataset are shown in Figure 5.7.

This domain is considered to be significantly more difficult than the

Omniglot domain since the images are less sparse and more complex in content. In fact, even for humans, it takes a trained eye to diagnosis what diseases are present in each image. Past approaches generally apply the classical MTL DNN architecture [152] and the current state-of-the-art approach uses a slightly modified version of Densenet [114], a widely used, hand-designed architecture that is competitive with the state-of-the-art on the Imagenet domain [57]. The images are divided into 70% training, 10% validation, and 20% test while the metric used to evaluate the performance of the network is the average area under the ROC curve for all the tasks (AUROC). Although the actual images are larger, all existing approaches preprocess the images to be 224×224 pixels, the same size used by many Imagenet DNN architectures.

5.6.2 Setup for Chest X-ray Domain

For the Chest X-ray experiments, only the CMSR variant of Codeep-NEAT was evaluated because it has been show to perform better than CM in the Omniglot domain. One major technical challenge faced was that evolving networks with a separate set of soft-ordering weights for each task results in much longer training times than in the Omniglot domain. This increase in training time was due to how soft-ordering layers are implemented in the deep learning library being used and the significant increase of the dataset size when compared to Omniglot. As a result, an optimization was introduced where the same set of Soft-ordering weights are used for each of the tasks or diseases. Preliminary experiment results show that the optimization did not

Name	Description
Normal	This is the same search space that has been used for the Omniglot evolution experiments. It covers arbitrary configurations of soft-ordering architectures.
Expanded	Compared to normal, the space is expanded by increasing the number of filters, module output filter size, and module output activation functions that can be discovered by evolution.
Hypercolumn	Same search space as normal, however the input is not the image directly. Instead, the image is given as input to Densenet and the feature maps from various key layers (also known as hypercolumns [48]) in Densenet are used as input to the evolved network instead.

Table 5.2: Summary of different search spaces; for normal and expanded, the input size is 224×224. For encoder, the input size is 28×28.

significantly hurt performance, but allows very significant speedups during training.

CMSR was used to evolve networks in different architecture search spaces. The search spaces from the worst to the best performance are normal, expanded, and hypercolumn and are described in detail in Table 5.2. Out of the three, the hypercolumn search space is the smallest while expanded is the largest, with normal somewhere in between the two.

The configuration for CMSR when evolving networks in the Chest X-ray domain is similar to that for Omniglot. However, there are some differences. For CMSR fitness evaluations, all networks were trained using Adam [70] for 8 epochs. After training is completed, AUROC was computed over all images in the validation set and returned as the fitness. No data augmentation was performed during training and evaluation in evolution, but the images were

normalized using the mean and variance statistics from the Imagenet dataset.

As in the Omniglot domain, CMSR was initialized with approximately 50 modules (in four species) and 20 blueprints (in one species). During each generation, 100 networks were assembled from modules and/or blueprints for evaluation. Random initialization of the blueprints and modules was disabled since evaluations were found to be too slow in the first few generations. The evaluation of assembled networks was distributed over 100 machines, each equipped with a GTX 1080 GPU. The average time for training was usually around 3-4 hours depending on the network size, although for some larger networks the training time exceeded 12 hours.

After evolution converged, the best evolved network was trained for an increased number of epochs using the ADAM optimizer [70]. Similar to other approaches to neural architecture search [119, 169], the model augmentation method was used, where the number of filters of each convolutional layer was increased. Data augmentation was also applied to the images during every epoch of training, including random horizontal flips, translations, and rotations. The learning rate was dynamically adjusted downward based on the validation AUROC every epoch and sometimes reset back to its original value if the validation performance plateaued. After training was complete, the test images were evaluated 20 times with data augmentation enabled and the model outputs were averaged to form the final prediction result.

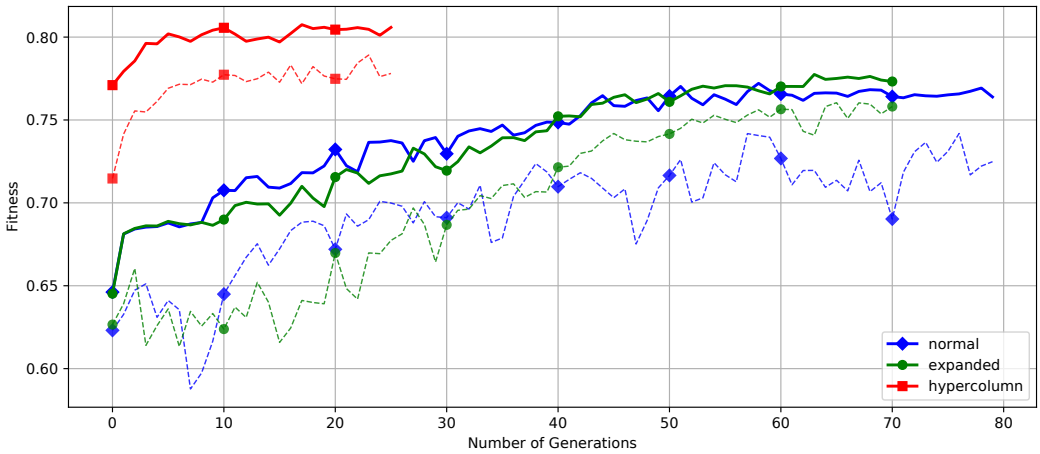


Figure 5.8: Comparison of fitness over generations of CMSR with different search spaces as described in Table 5.2. Solid lines show the fitness of best assembled network and dotted lines shows the mean fitness. The hypercolumn search space is quickest to converge and reaches the best fitness.

5.6.3 Results for Chest X-ray Domain

Figure 5.8 summarizes how the mean and best fitness improved during evolution for the three search spaces mentioned in the Table 5.2. The hypercolumn search space is the smallest and as a result it converged first. Such behavior is expected since hypercolumn evolved a smaller network to process high level features extracted from the larger, more powerful Densenet network. Next to converge is the normal search space, which evolved networks from scratch, but with less free hyperparameters than the expanded search space. The expanded search space has the largest number of different architecture configurations to explore and thus was the slowest to converge. Overall, the hypercolumn search space achieved the best fitness as it was able

Algorithm	Test AUROC (%)
1. Wang et al. (2017) [152]	73.8
2. CheXNet (2017) [114]	84.4
3. Google AutoML (2018) [12]	79.7
4. Normal (CMSR)	81.1
5. Expanded (CMSR)	82.2
6. Hypercolumn (CMSR)	84.3

Table 5.3: AUROC on test set for existing approaches that use hand-designed architectures and networks which are evolved using CMSR. CMSR combined with the hypercolumn approach results in an architecture that is competitive with the state-of-the-art.

to make use of higher level features extracted another powerful state-of-the-art network.

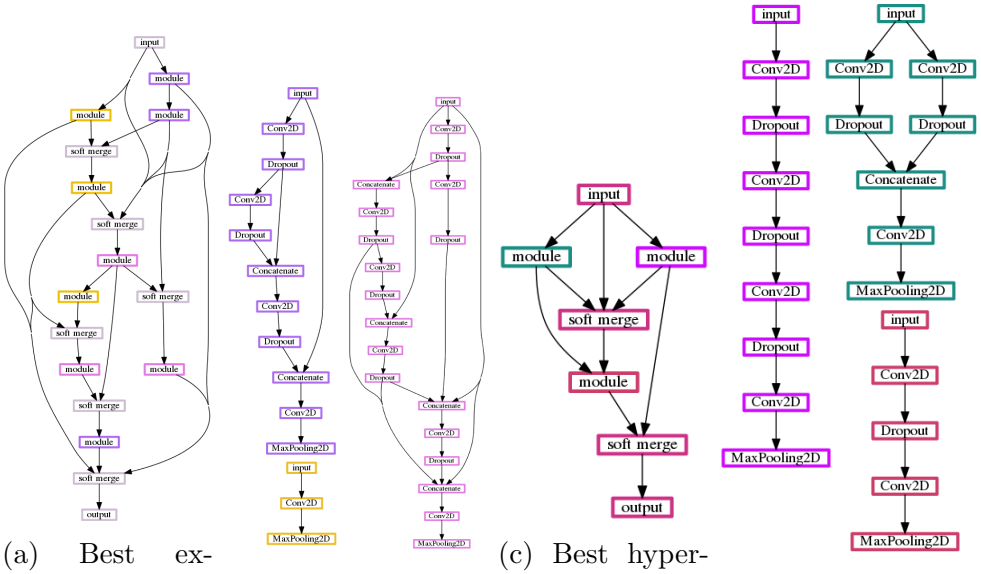
Table 5.3 compares the performance of the best evolved networks with existing approaches that use hand-designed network architectures on a holdout test set of images. These include results from the authors who originally introduced the Chest X-ray dataset and also CheXNet [114], which is the currently published state-of-the-art. For comparison with existing architecture search algorithms, results from Google AutoML [12], a service for automatically optimizing the architecture and weights of the network given a dataset, are also listed. The results show that while CheXNet overall has the highest AUROC score, the results from evolved networks in the hypercolumn search space are almost equivalent. More impressively, the results from evolving the network architectures from scratch (normal and expanded) are only around 2 AUROC points behind the state-of-the-art. Lastly, all the CMSR results exceed the

performance of the model generated by Google AutoML.

Visualizations of the best evolved networks from the expanded and hypercolumn search spaces are displayed in Figure 5.9. The visualization for the normal search space has been omitted because it looks very similar to the best evolved network from the expanded search space. For the expanded network, Figure 5.9a shows the routing between the modules that is evolved using CMSR while Figure 5.9b shows the architecture of different modules themselves. Figures 5.9c and 5.9d show the same, but for the hypercolumn network. Each unique module architecture is associated with a corresponding color. As in the Omniglot domain, the evolved topologies show a mixture of both complex and simple modules. However, it is interesting to note that the evolved architecture for the hypercolumn search space is significantly simpler than for the expanded one. The hypercolumn network is given high-level features as input and thus fewer layers and modules are required to process the input.

5.7 Conclusion

This chapter has introduced an extension to CoDeepNEAT (CMSR) that generalizes the soft-ordering architecture and allows CoDeepNEAT to evolve more powerful networks in the MTL domain. Promising experiment results are shown for two MTL domains, namely the Omniglot character recognition and the Chest X-ray image classification domain. The results support that CMSR is a powerful approach and is capable of evolving multitask networks



(a) Best expanded module routing. (b) Best expanded modules. (c) Best hypercolumn module routing. (d) Best hypercolumn modules.

Figure 5.9: Overview of the best architectures evolved for the expanded (Figures 5.9a and 5.9b) and hypercolumn (Figures 5.9c and 5.9d) search spaces. The best hypercolumn network is significantly simpler than the best expanded network.

that achieve results capable of meeting or exceeding the state-of-the-art.

Chapter 6

Multiobjective Coevolution of Deep Neural Network Architectures

This chapter will describe how multiobjective optimization can be applied to architecture search for deep neural networks. The foundations and related work of evolutionary multiobjective optimization are covered in Section 2.2.5. This chapter will first review the motivations and two interesting use cases for multiobjective neural architecture search. Second, an overview is given of how CoDeepNEAT, a single-objective optimization algorithm, is modified to support multiobjective search. Third, multiobjective CoDeepNEAT is combined with novelty search to accelerate performance during evolution and experimental results are presented in the image captioning domain. Fourth, multiobjective evolution is combined with CMSR to optimize both the fitness and complexity and experimental results are reported in the Chest X-ray multitask classification domain.

6.1 Motivation

Multiobjective optimization techniques are widely used in the neuroevolution and reinforcement learning domain. For example, it has been used to

evolve diverse multimodal behaviors for agents in games [127] and also has been applied to help escape deceptive local optima in the fitness landscape [82,130]. Unfortunately, there has not been much work until very recently [35,93] on applying multiobjective techniques to neural architecture search. However, there are many scenarios where having another objective besides the performance of the network could be beneficial. This chapter addresses two such use cases:

Accelerating convergence by overcoming deception in architecture search. One of the challenges of neural architecture search is that the fitness landscape is very ill-defined and there many deceptive traps or local optima in the search space. For example, in the ImageNet image classification domain, ResNet [51] is a local optima while GoogLeNet [145] is another one. Although the performance between the two networks are relatively close, their network architectures can not be further apart. This observation suggests that the fitness landscape is highly deceptive and there is no obvious path going from one good architecture to another good one. Fortunately, novelty search [80] is a promising method for overcoming deception during optimization. It would be useful if novelty and fitness based objectives can be combined to help escape local optima more easily, thus accelerating convergence to good network architectures during evolution.

Optimizing DNNs for Mobile Applications. Smartphones are becoming increasingly prevalent and replacing other devices such as cameras or music players. AI enabled smartphone applications are also becoming more popular but they often are restricted by the limited memory present in mobile devices.

As DNNs can have up to hundreds of millions of weights, they often cannot fit inside the one or two gigabytes of RAM that most smartphones have. Much research is focused on minimizing the size of a network while maximizing its performance [56]. Multiobjective architecture search is a promising approach for automatically discovering compact network architectures that also perform well on a benchmark or task.

6.2 Multiobjective CoDeepNEAT

In this section, CoDeepNEAT is modified and extended to support multiobjective optimization. The procedure for integrating novelty search into multiobjective CoDeepNEAT is also covered in more detail.

6.2.1 Algorithm Overview

In a single-objective evolutionary algorithm like CoDeepNEAT, evolutionary elitism is applied in both the blueprint and the module populations. The elitism involves preserving the top fraction F_l of the individuals within each species into the next generation based on their ranking within the species. Normally, this ranking is based on sorting the individuals by the primary and only objective fitness. In the multiobjective version of CoDeepNEAT (MCDN), this ranking is computing using multiple fitness values for each individual. The ranking is based on generating successive Pareto fronts [29, 168] from the individuals and ordering the individuals within each Pareto front based on either the primary objective fitness or a secondary objective value. This approach

to ranking the individuals with Pareto fronts draws inspiration from NSGA-II [31], a powerful and widely used multiobjective EA.

In addition, elitism is applied at the coevolutionary level where networks are assembled together from the blueprints and modules. In particular, elitism is used to determine the assembled networks that are preserved and reevaluated in the next generation. The ranking method that is used to determine what fraction F_u assembled networks are preserved is the same as the method used for the blueprints and modules.

While the ranking method can be generalized to any number of objectives, the implementation inside CoDeepNEAT is limited to two objectives. This restriction is for the sake of simplicity and because the use cases described earlier do not require more than two objectives. Algorithm 7 gives a detailed explanation of how the ranking is performed within a single generation of MCDN for the blueprints and modules. Similarly, Algorithm 8 details how the assembled networks are ranked. Algorithm 9 shows how the Pareto front, which is necessary for ranking, is calculated given a group of individuals that have been evaluated for each objective.

There is also an optional configuration parameter for MCDN (last line in Algorithm 8) that allows the individuals within each Pareto front to be sorted and ranked with respect to the secondary objective instead of the primary one. This configuration parameter can control whether the primary or secondary objective is favored more during evolution. The right choice for this configuration parameter will depend on the use case.

Algorithm 7 Ranking for Blueprints/Modules in MCDN

1. Given population of modules/blueprints, evaluated primary and secondary objectives (X and Y)
2. For each species S_i during every generation:
  3. Create new empty species $\hat{S}_i$
  4. While S_i is not empty
    5. Determine Pareto front of S_i by passing X_i and Y_i to Alg. 9
    6. Remove individuals in Pareto front of S_i and add to $\hat{S}_i$
  7. Replace S_i with $\hat{S}_i$
  8. Truncate $\hat{S}_i$ by removing the last fraction F_l
  8. Generate new individuals using mutation/crossover
  9. Add new individuals to $\hat{S}_i$, proceed as normal

Algorithm 8 Ranking for Assembled Networks in MCDN

1. For each generation:
  2. Create archive A using leftover networks from last generation
  3. Add to A newly assembled networks
  4. Evaluate all networks in A on each objective O_i
  5. Create new empty archive $\hat{A}$
  6. While A is not empty
    7. Determine Pareto front of A using Alg. 9
    8. Remove individuals in Pareto front of A and add to $\hat{P}$
  9. Replace A with $\hat{A}$
  10. Truncate $\hat{A}$ by removing the last fraction F_u

Algorithm 9 Calculating Pareto front for MCDN

1. Given list of individuals I, and corresponding objective fitnesses X and Y
2. Sort I in descending order by first objective fitnesses X
3. Create new Pareto front PF with first individual I_0
4. For each individual I_i, $i > 0$
  5. If Y_i is greater than the Y_j, where I_j is last individual in PF
    6. Append I_i to PF
7. Sort PF in descending order by second objective Y (Optional)

Figure 6.1: Overview of lower and upper levels of MCDN perform ranking of individuals and also how the Pareto front is calculated from two objectives.

6.2.2 Combining Multiobjective CoDeepNEAT with Novelty Search

Novelty search is a powerful method for overcoming deceptive traps in evolution and local optima in the fitness landscape [80]. It avoids deceptive local optima by pressuring the EA to search for solutions that result in novel phenotypical behaviors and encourage exploration of the search space. The foundations of novelty search and how the novelty of an individual is calculated are reviewed in Section 2.2.6. As in previous work [82,130,156], novelty search is combined with MCDN by using it as a secondary objective in addition to the performance of the network, which is still the primary objective.

In architecture search, the novelty behavior metric is not as well defined as in reinforcement learning domain. The evolved network is not interacting with an environment where there is an easily observable behavior [80]. Thus, the behavior metric is instead defined by extracting features or embeddings from the graph structure of the evolved networks, as Figure 6.2 shows.

During evaluation of a candidate network, these features are computed at the end of training and are then concatenated into a 11-dimensional long behavior vector. The vector is then added to a common shared novelty archive, which is used to compute the novelty value for the network by calculating its Euclidean distance to the closest other network in behavior space. The novelty score and fitness for the network are then returned back to MCDN. As the fitness is the only metric that truly matters in the end, the parameter described earlier is configured to favor the primary objective (fitness).

1. Number of parameters of the network
2. Number of layers in the network (N)
3. Number of connections between layers (E)
4. The length of the longest path within the network
5. The density of the network, defined as $D = \frac{|E|}{|N|(|N|-1)}$
6. Maximum of the number of connections for each layer
7. Mean of the number of connections for each layer
8. Standard deviation of the number of connections for each layer
9. Maximum of the pagerank for each layer
10. Mean of the pagerank for each layer
11. Standard deviation of the pagerank for each layer

Figure 6.2: List of the hand-crafted features used to characterize the behavior of each evolved network for novelty search. The novelty score generated using the behavior metric is used as a secondary objective along with fitness.

6.2.3 Experimental Results in MSCOCO Image Captioning Domain

This section reviews results from applying MCDN to the MSCOCO image captioning domain. The domain and the metrics used to characterize network performance are presented in Section 4.5.

The primary objective fitness is the performance relative to a baseline image captioning architecture [150] on common sentence similarity metrics such as BLEU, METEOR, and CIDER. Figure 6.3 shows how fitness improved during evolution for both MCDN and CoDeepNEAT. It is interesting to note that MCDN improved slower than CoDeepNEAT for the first 12 generations. This is expected since MCDN used both novelty and fitness as the objectives

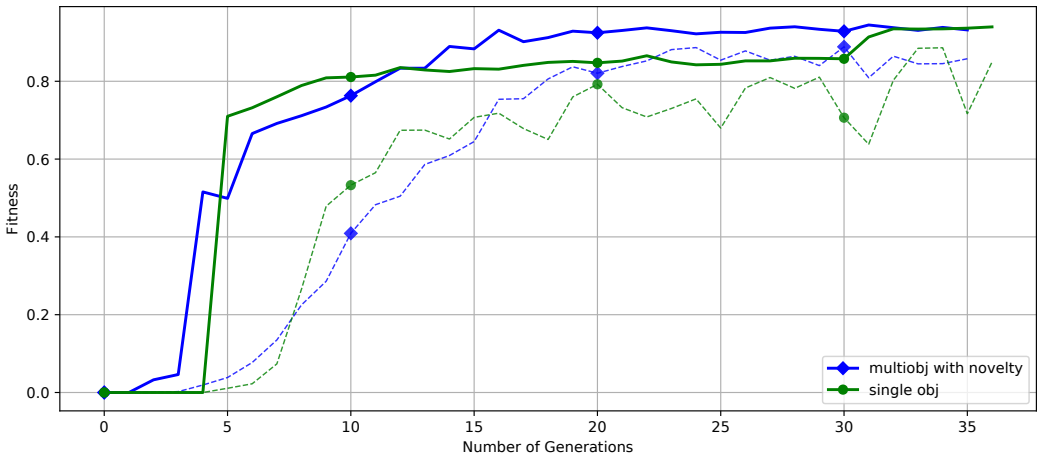
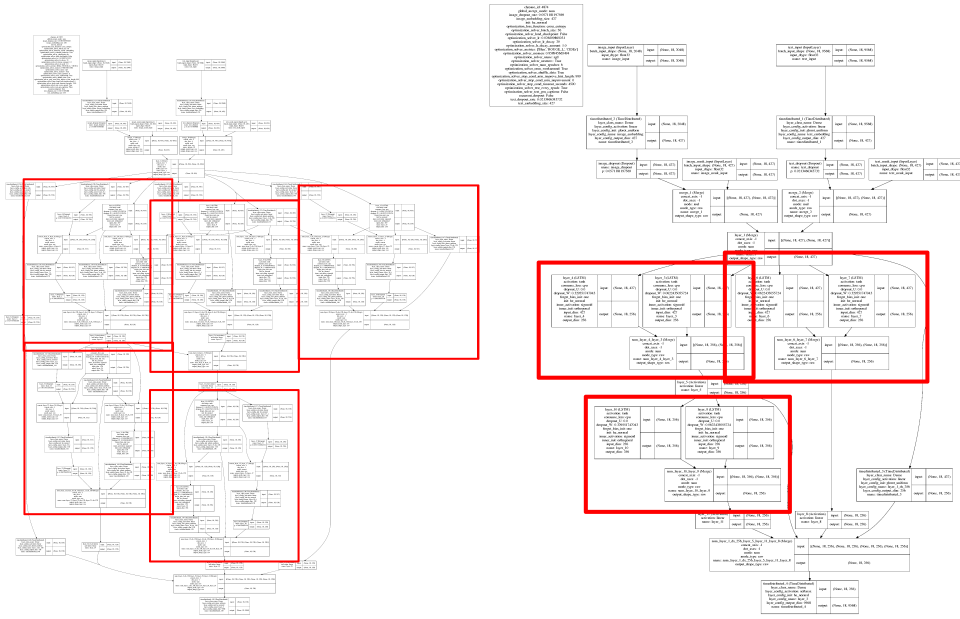


Figure 6.3: Comparison of fitness over generations of MCDN (multiobjective) and CoDeepNEAT (single-objective) on the MSCOCO image captioning domain. Solid lines show the fitness of best assembled network and dotted lines shows the mean fitness. MCDN is able to converge at generation 15, 15 generations faster than CoDeepNEAT.

and novelty required some time to build up an archive of individuals to be able to accurately calculate distances of individuals in behavior space. However after 12 generations, MCDN began to outperform CoDeepNEAT, and was able to converge roughly 15 generations before CoDeepNEAT. Novelty encouraged exploration of the architecture search space and allowed MCDN to overcome deceptive traps in network design. Both the multiobjective MCDN and single CoDeepNEAT eventually converged to around the same fitness ceiling. This fitness ceiling does not represent the final performance of evolved networks and was due to the reduced number of training epochs used during evolution to keep the time of each generation reasonable.



(a) Best network for MCDN (b) Best network for CoDeepNEAT

Figure 6.4: Visualizations of best networks evolved by MCDN (Figure 6.4a) and CoDeepNEAT (Figure 6.4b) on the image captioning domain. The modules in both networks are highlighted in red. Both networks are able to reach similar fitness after six epochs of training, but the network evolved by MCDN is significantly more complex and contains more novel module structures.

Figure 6.4a shows the architecture of the best evolved network by MCDN while Figure 6.4b shows the best architecture of the best evolved network by CoDeepNEAT. Both algorithms were run for approximately the same number of generations and with the same evolution parameters (except for those related to multiobjective search) but the networks discovered are very different. Interestingly, the network evolved by MCDN is significantly more complex with regards to the number of layers, modules, and the depth of the network. The increased complexity could suggest that novelty is helping evolution escape local optima in fitness that are caused by the networks being not deep enough.

6.3 Multiobjective CMSR

Multiobjective CMSR (MCMSR) is an immediate extension of CMSR to multiobjective search by incorporating MCDN's Pareto front and ranking algorithms (described in Section 6.2).

6.3.1 Using Multiobjective CMSR for Network Complexity Minimization

MCMSR is applied to simultaneously optimize the evolved network architectures and to minimize the complexity of the network. There are many ways to characterize how complex a DNN is, including number of parameters, number of floating point operations (FLOPS), and training/inference time of the network. The most commonly used metric is number of parame-

ters because other metrics can change depending on the deep learning library implementation and performance of the hardware. In addition, this metric is becoming increasingly important in mobile applications [56] as mobile devices are highly constrained in terms of memory and require networks with as high performance per parameter ratio as possible. Thus, number of parameters is also used as the secondary objective with MCMSR in the experiments in the following section.

Although the number of parameters can be very large it does not pose a problem for MCMSR since it only cares about the relative rankings between different objective values, not their absolute differences. As a result, no scaling of the secondary objective is required for MCMSR. Preliminary results show that favoring the secondary objective (minimizing network complexity) during ranking will lead to the evolution of slightly smaller networks with slightly worse fitness. Since network performance is typically more important that achieving the smallest possible networks in mobile applications, the configuration parameter described in Section 6.2 is set to put more pressure on optimizing the primary objective (fitness).

An alternative approach for minimizing the network size is to add a complexity penalty term to the fitness for single objective CMSR. However this approach only discovers a single kind of trade-off between the size and performance of the evolved networks, while MCMSR can explore more varied trade-offs between the two objectives. The complexity penalty also introduces a new hyperparameter which must be tuned beforehand and preliminary ex-

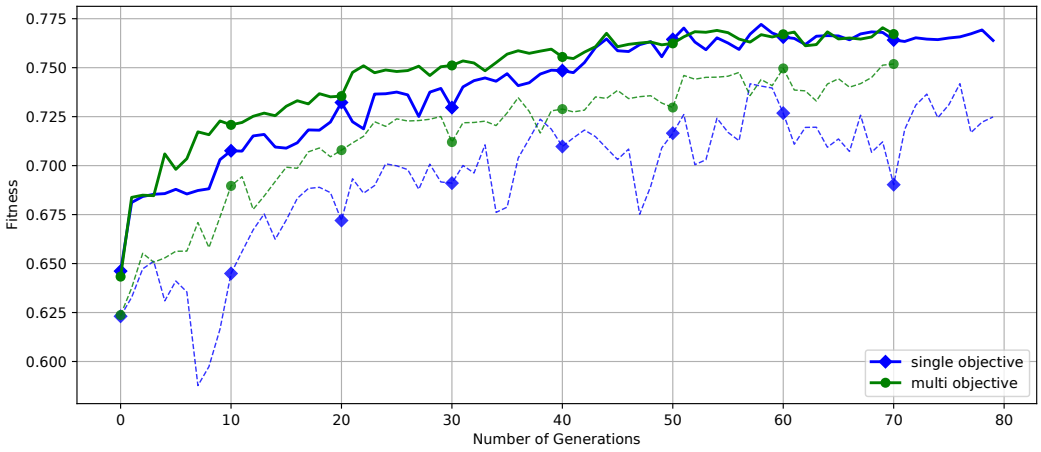


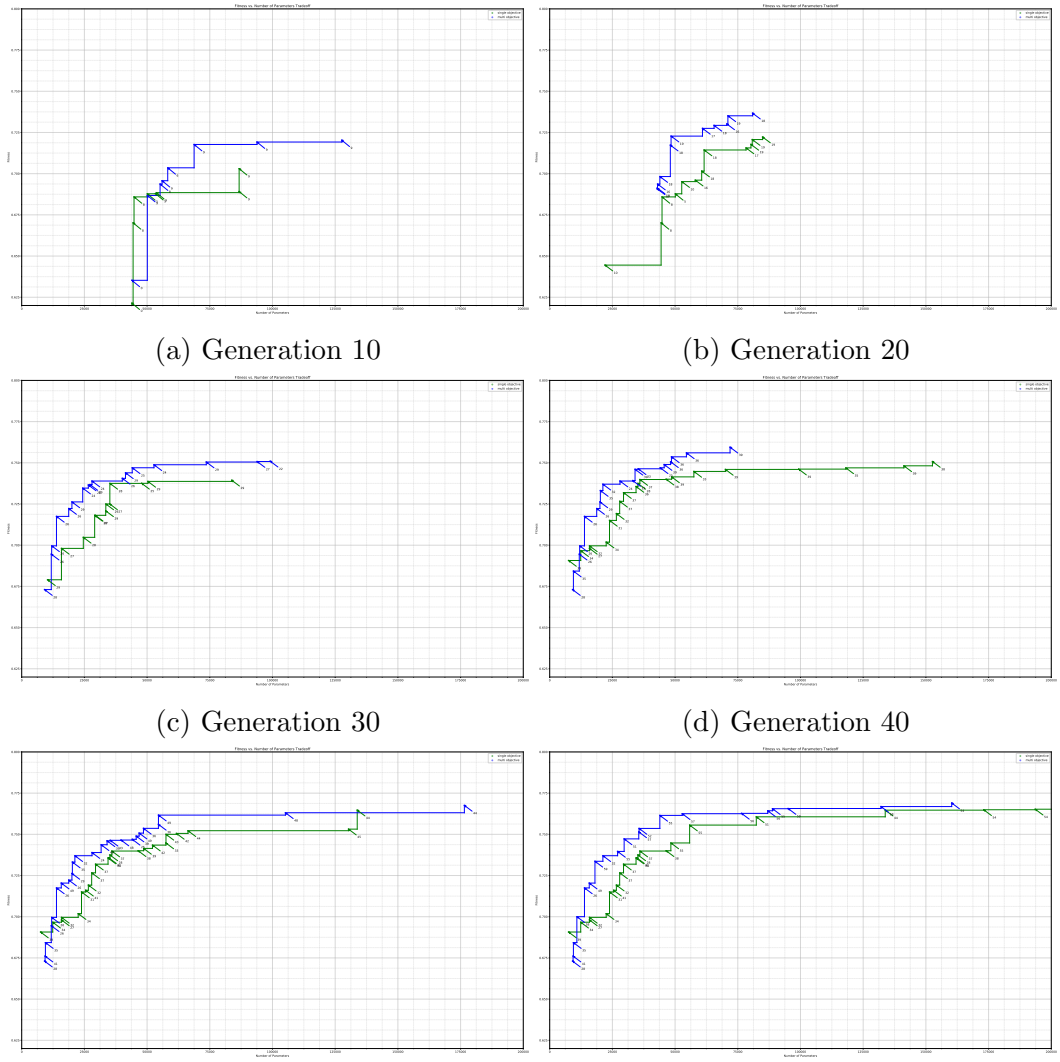
Figure 6.5: Comparison of fitness over generations of the single-objective CMSR and multiobjective MCMSR with the normal search space (Table 5.2). Solid lines show the fitness of best assembled network and dotted lines shows the mean fitness. Minimizing network complexity also seems to benefit the primary objective; MCMSR is able to achieve higher fitness and converge faster than CMSR.

periments showed that applying such a penalty often caused evolution to prematurely converge due to lack of diversity in the population.

6.3.2 Experimental Results in Chest X-ray domain

MCMSR, the multiobjective version of CMSR, was evaluated on the Chest X-ray domain, described back in to Section 5.6.1. The experimental setup for MCMSR is very similar to that of CMSR, which is detailed in Section 5.6.2. The primary objective fitness for MCMSR was still AUROC, and the secondary objective was the number of parameters in the evolved network. The search space was the normal search space described in Table 5.2; the exper-

imental configuration parameters like population size and training time during evaluation remained the same. After evolution, the best evolved network from MCMSR was trained fully using the same methods in Section 5.6.2. The same data augmentation techniques (random translations, flips, and rotations) and model augmentation method [119, 169] were used.



(a) Generation 10

(b) Generation 20

(c) Generation 30

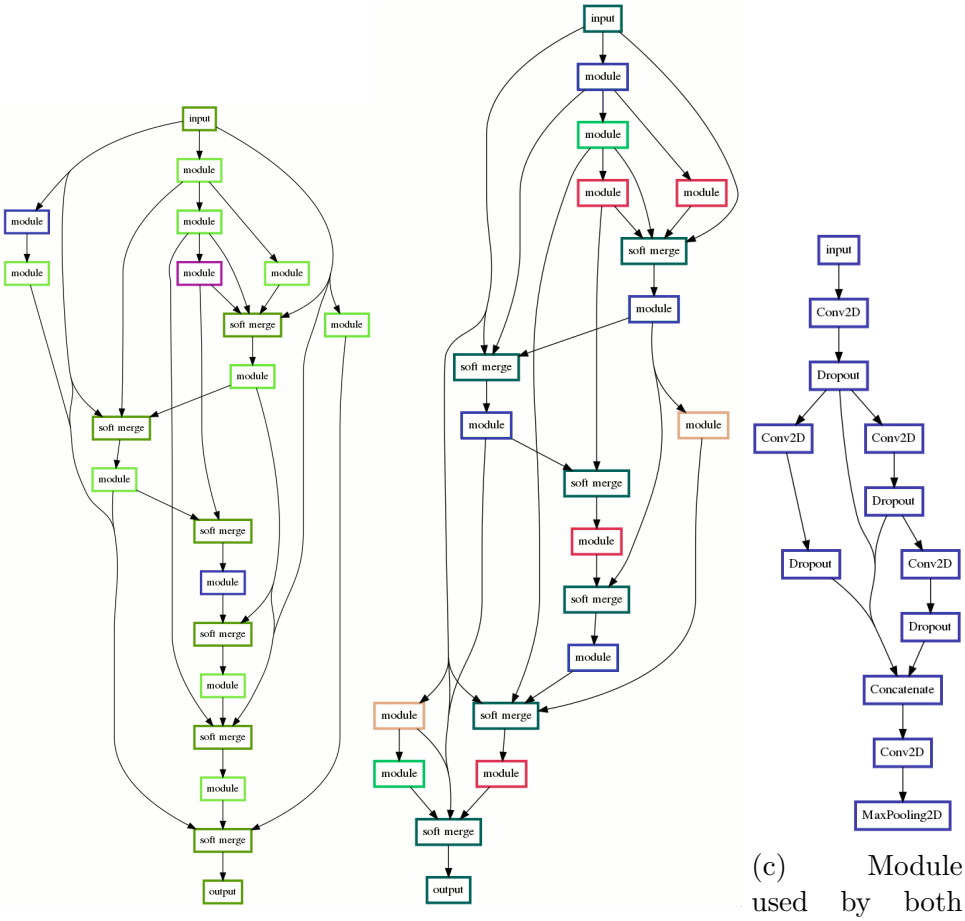
(d) Generation 40

(e) Generation 50

(f) Generation 60

Figure 6.6: Comparison of the single and multiobjective Pareto fronts for CMSR (green) and MCMSR (blue) respectively at various generations during evolution. The X-axis shows number of parameters (secondary objective) while Y-axis shows AUROC fitness (primary objective). The Pareto front for MCMSR consistently dominates over CMSR's Pareto front. In other words, MCMSR discovers trade-offs between complexity and performance that are always better than those found by CMSR.

Figure 6.5 compares the main objective fitness (AUROC) during evolution for CMSR and MCMSR. Although MCMSR is optimizing for both the primary fitness and the complexity of the network, it is able to discover better performing networks faster. Both methods eventually were able to reach the same fitness, but MCMSR converges faster than CMSR. In addition, as expected from an multiobjective optimization algorithm, MCMSR is able to discover networks with fewer parameters. As shown in Figure 6.6, the Pareto front generated during evolution by MCMSR (blue) dominates that of the CMSR (green) when compared at same generations during evolution. Although CMSR is a single-objective algorithm, it is still possible to generate a Pareto front by giving the primary and secondary objective values of all the networks discovered in past generations to Algorithm 9. The Pareto front for MCMSR was also created in a similar manner.



(a) 56K network (b) 125K network (c) Module used by both networks

Figure 6.7: Visualizations of networks with different complexity discovered by MCMSR. The performance of the significantly smaller 56K network (Figure 6.7a) is nearly as good as that of the larger 125K network (Figure 6.7b). The smaller network uses only two instances of the module architecture shown in Figure 6.7c while the larger network uses four instances of the same module. These two networks show that MCMSR is able to find good trade-offs between two conflicting objectives by clever usage of modules.

Visualizations of the some of the networks evolved by MCMSR are

shown in Figure 6.7. MCMSR was able to discover a very powerful network (Figure 6.7b) that achieved 77 AUROC after 8 epochs of training. This network has 125K parameters and is already significantly smaller than networks with similar fitness discovered with CMSR. Furthermore, MCMSR was able to discover an even smaller network (Figure 6.7a) with only 56K parameters and a fitness of 74 AUROC after 8 epochs of training. The main difference between the smaller and larger network is that the smaller one uses a particularly complex module architecture (Figure 6.7c) only two times within its blueprint while the larger network uses the same module four times.

6.4 Conclusion

This chapter introduced a multiobjective extension of CoDeepNEAT and its multitask variant (MCDN, MCMSR). MCDN was combined with novelty search and was applied to the MSCOCO image captioning domain. Novelty search allowed MCDN to escape deception in the architecture search space and as a result, MCDN was able to reach the best possible fitness faster than CoDeepNEAT. Likewise, MCMSR was applied to the Chest X-ray domain to optimize both the complexity and the fitness of the networks. MCMSR was consistently able to discover networks with better trade-offs with respect to complexity and fitness when compared to CMSR. These results validate the effectiveness of multiobjective optimization and its relevance to neural architecture search.

Chapter 7

Discussion and Future Work

While experimental results in many domains such as image classification, image captioning, and multitask learning have validated the power of the methods described in this dissertation, it is still necessary to draw some higher level conclusions regarding evolutionary approaches to neural architecture search in general. In order do to so, this chapter will examine the algorithms described in this dissertation in more detail. By discussing their successes limitations, promising areas of future work to improve performance are revealed.

7.1 DeepNEAT and CoDeepNEAT

This section summarizes the important lessons learned from the experiments that used DeepNEAT or CoDeepNEAT evolve the network architecture. Future work for improving CoDeepNEAT, also applicable to the multitask and multiobjective versions such as CMSR, MCDN and MCMSR are proposed as well.

7.1.1 Discussion

The results for DeepNEAT and CoDeepNEAT show that the evolutionary approach to optimizing deep neural networks is feasible: The results are comparable to hand-designed architectures in benchmark tasks, and it is possible to build real-world applications based on the approach. It is important to note that the approach has not yet been pushed to its full potential. It takes a couple of days to train each deep neural network on a state-of-the-art GPU, and over the course of evolution, thousands of them need to be trained. Therefore, the results are limited by the available computational power. Interestingly, since it was necessary to train networks only partially when evaluating them, evolution is biased towards discovering fast learners instead of top performers. This is an interesting result on its own since evolution can be guided with goals other than simply accuracy, including training time, execution time, or memory requirements of the network.

Significantly more computational resources are likely to become available in the near future. Already cloud-based services such as Amazon AWS [1] offer GPU computation with a reasonable cost, and efforts to harness idle cycles on gaming center GPUs are underway. At Sentient Technologies [7], where most of the research in this dissertation was done, a distributed AI computing system called DarkCycle was built between 2015 and 2017. Darkcycle could potentially utilize idle cycles of 2M CPUs and 5000 GPUs around the world, resulting in a peak performance of 9 petaflops, on par with the fastest supercomputers in the world. Not many approaches can take advantage of such

power, but evolution of deep learning neural networks can. The search space of different components and topologies can be extended, and more hyperparameters be optimized. Given the results in this paper, this approach is likely to discover designs that are superior to those that can be developed by hand today. It is also likely to make it possible to apply deep learning to a wider array of tasks and applications in the future.

DeepNEAT and CoDeepNEAT (along with its variants such as CMSR, CMDN, and MCMSR) are automated and efficient methods for DNN architecture and hyperparameter optimization that is applicable to many domains and use cases. However there are still many opportunities to improve the performance and search space explored. For example, recent innovations in the architecture and training procedure of DNNs such as attention layers [147] and cyclical learning rates [134] could be incorporated into the architecture search space.

There also exists much potential for improving the running time of evolution. The experiments described in Chapter 4, 5 and 6 are expensive to run, requiring more than 100 worker machines each equipped with a GPU. Another major compromise with the current version of CoDeepNEAT is that during evaluation, the number of training epochs for the networks are capped at a very low number for the sake of limiting generation time to a few hours and keeping the overall computational time tractable. Unfortunately, lack of training could lead to biased results and premature convergence of evolution to a suboptimal solution. A possible way to deal with this problem might be

to learn a regression model to predict the fitness when the network is fully trained.

7.1.2 Future Work

Since DeepNEAT has already been superseded and outperformed by CoDeepNEAT, most of the focus of future work will be improving CoDeepNEAT and its variants. There are two major areas where there is potential for improvement: (1) Generalizing and enlarging the search space that CoDeepNEAT explores, thereby making it more likely to discover novel architectures that beat existing hand-designed ones. (2) Improving the efficiency of evolution: For a given computational and time budget, maximize the number of candidate architectures to be evaluated. In other words, if the amount of time to find a state-of-the-art DNN is minimized, even better networks can be found by running evolution longer.

Potential ideas for achieving the first goal include the following:

Layer-type-aware mutation and crossover. In the current implementation of CoDeepNEAT, the neural network layer type for each node in the chromosome is stored as part of the binary hyperparameter table. Consequently, during evolution, CoDeepNEAT is not aware of which layer corresponds to which node and thus during mutation and crossover, might cause incompatible layers to be grouped together (for example a convolutional and a recurrent layer). In the current implementation, this drawback is avoided by restricting evolution to just a few layer types that are guaranteed to be compatible with

each other. However, in order increase the search space and diversity of architectures evolved, explicit support for many different types of layers must be added to CoDeepNEAT.

One solution is to store the layer type explicitly and separately from the rest of the hyperparameters and to maintain a compatibility matrix that indicates which pairs of layers are allowed to be connected with each other and which are not. During mutation, all connection genes in the chromosome are checked to ensure no layer type violations; if any violations are discovered, the mutation is declared invalid and aborted. Similarly, the crossover operation will be aborted if it leads to two incompatible layers connected to each other. Lastly, depending on the domain, the user should be able to specify to the EA a set of relevant layer types to be used during evolution.

Support for multiple input and output types. One issue with the current implementation is that there is only one input and one output layer in each evolved DNN architecture. While this limitation is fine for tasks such as image classification, other domains such as MTL could benefit from architectures with multiple input and output connections. Such an extension is straightforward for DeepNEAT (since NEAT already supports multiple input and output neurons), but the extension to CoDeepNEAT requires more effort. For example, should only the blueprint chromosomes support multiple inputs and outputs or should multiple inputs be handled in the modules instead? One issue with modules having multiple inputs and outputs is that connecting two modules together will become more difficult. There now could be many

different connectivity patterns between the output layers of one module and the inputs of another module. For example, each output layer might connect randomly to another input layer, or it could connect to every possible input. One of the possible future experiments will be to explore different connectivity patterns and evaluate how they affect the performance of CoDeepNEAT.

Evolution of Custom Layer Types and Loss Functions. While CoDeepNEAT can evolve how different layer types connect, it is unable to optimize the internals of a layer. Each layer in a neural network can be thought of as a mathematical function, in other words, a hierarchical composition of different types of elementary mathematical operations such as products, sums, etc. Such tree-like structure can be clearly seen in the equations for describing an LSTM layer [55]:

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\ h_t &= o_t \odot \sigma_h(c_t) \end{aligned} \tag{7.1}$$

Evolving novel functions makes it possible to create entirely new layers with potentially better performance than existing ones. Since mathematical functions often have a tree-like structure, genetic programming (GP) [110,117,142] is likely to be more an appropriate approach. For example, GP and CoDeepNEAT can be setup to run together in parallel. From time to time, CoDeepNEAT can pull novel layers from the GP population pool, use them in candidate DNN architectures and then assign the performance of these DNN ar-

chitectures as fitness to these novel layers. A similar approach could also be applied to evolve the loss function that is used during training of candidate DNNs generated by CoDeepNEAT.

Evolution of Preprocessing and Data Augmentation. In many tasks, data augmentation and preprocessing have been shown to be crucial in achieving good performance, especially when the amount of training data is limited. The boost in performance from data augmentation sometimes can exceed the improvement achieved through a new architecture algorithm or a new training algorithm. Data augmentation can be thought of as a special non-differentiable and non-trainable computation graph (where each node is some sort of data transformation) that is applied to each training sample before it is passed as input into the DNN. Similarly, data preprocessing is a similar computation graph that is applied only once to the input data before training. These computational graphs could be evolved with CoDeepNEAT, either through a coevolution approach where there is a separate population of data augmentation/preprocessing graphs alongside the blueprint/module populations, or as a special type of module that represents the input blueprint node. As with the layer types, it is necessary to check that incompatible data transformations are not placed right next to each other in the computational graph.

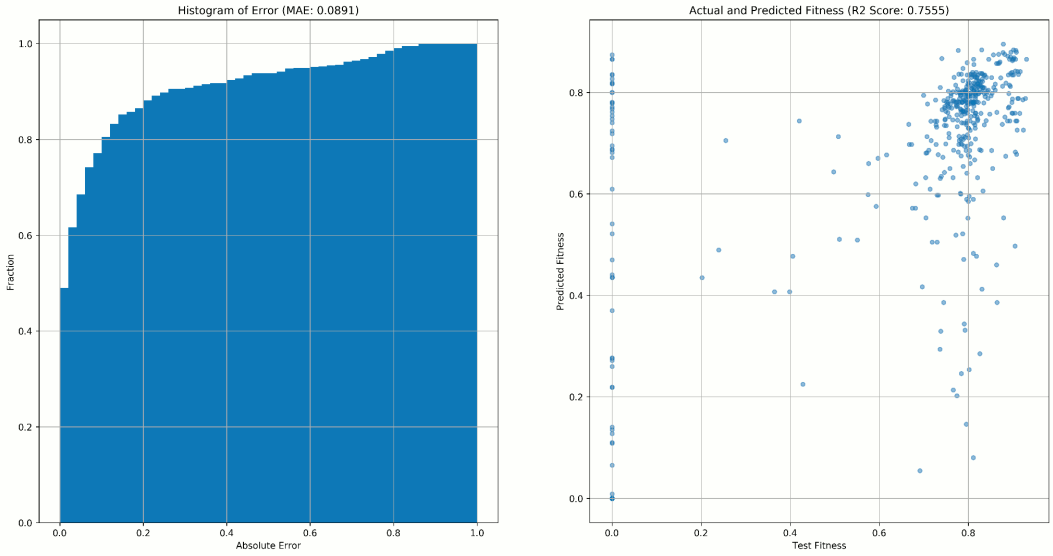
Promising ideas for achieving the second goal, improving the efficiency and performance of CoDeepNEAT, include:

Surrogate Optimization with Vector Representations of DNNs. As past work in bilevel optimization of control tasks [86] has shown, surrogate opti-

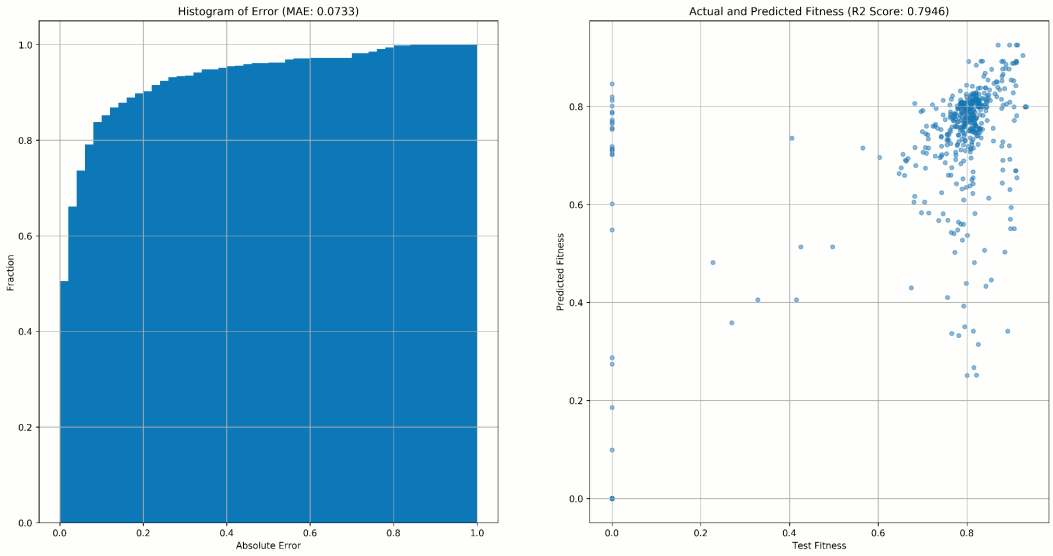
mization is an efficient way to reduce the number of evaluations required to reach a high fitness. In surrogate optimization, real fitness evaluations are replaced by predictions from a regression (surrogate) model trained on data collected by actual evaluations. There are two possible ways in which surrogate optimization can be used to improve the performance of CoDeepNEAT: (1) During mutation, a large number of mutated child chromosomes are generated. The networks that correspond to these chromosomes are assigned a fitness by the surrogate model and only those child networks with the highest predicted fitnesses are allowed to continue onto the next generation. This modification results in more intelligent mutations than a purely random approach. (2) During evaluation of individuals in the population, some of the individuals are randomly chosen to be evaluated by the surrogate model instead, thus reducing the burden for computationally intensive evaluation and training. The surrogate model is updated at the end of every generation using fitness and performance metrics collected from DNNs that underwent actual fitness evaluation.

One issue with using a regression model as the surrogate is that it requires a fixed length vector or feature representation of the network. As evolution of DNNs creates networks of arbitrary size, complexity, and structure, it is not easy to compress the network into a fixed length description. There has been recent work on using data-driven methods such as Deepwalk to come up with vector embeddings for nodes in graphs [111]. In Deepwalk, random walks across the nodes of the graph are used to generate embeddings

for each node through an embedding algorithm such as Word2Vec [101]. A extension of this idea called Skip-Graph [77] uses recurrent neural networks and random walks to generate a single embedding for the entire graph. As the topology of DNNs is essentially a graph, it will be relatively straightforward to extend the basic methodology behind Deepwalk and Skip-Graph to generating embeddings or representations of DNNs.



(a) hand-designed embedding



(b) Deepwalk embedding

Figure 7.1: Comparison of the predictive performance of hand-crafted embeddings described in Figure 6.2 and learned embeddings using Deepwalk [111]. The histogram of error is shown on the left while the correlation between predicted and actual fitnesses is shown on the right. Deepwalk is able to predict the fitnesses of the networks with smaller mean absolute error (MAE) and with higher correlation to the actual fitnesses than the hand-designed embeddings.

Preliminary experiments showed that Deepwalk can generate embeddings that perform better than handcrafted features such as those described in Figure 6.2. Figure 7.1 shows the performance of both in predicting the fitnesses of different network topologies in the MSCOCO image captioning domain. Deepwalk was first used to learn the embedding on a training set of about 3000 networks evolved from the MSCOCO experiments with CoDeepNEAT. Both approaches were then evaluated on a test set of 500 networks. As seen in Figure 7.1b, Deepwalk is able to learn a better embedding that predicts the fitness of the networks with less mean absolute error than the hand-designed embedding shown in Figure 7.1a (0.0733 vs 0.0891). In addition, the correlation between the predicted and actual fitness is higher for Deepwalk as well when compared to the hand-designed approach (0.7946 vs 0.7555). Future work will investigate the effectiveness of DNN embeddings or representations that are learned from data (such as Deepwalk and Skip-graph) during evolution, where they are used to train a surrogate model in an online manner.

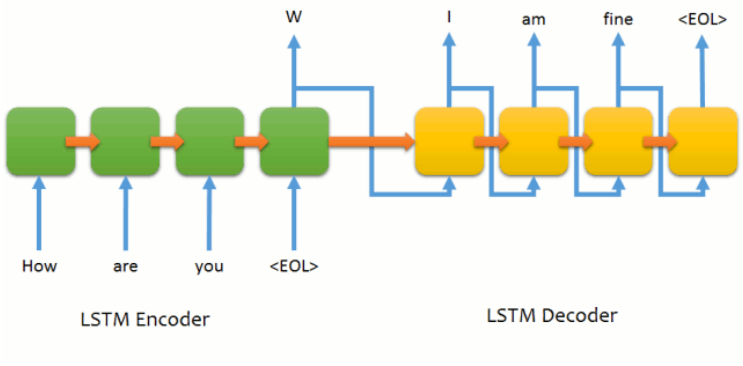


Figure 7.2: Visualization of a sequence-to-sequence model. The model is composed of two LSTM layers [8]. The LSTM on the left is called the encoder and the LSTM on the right is called the decoder. This sequence-to-sequence network can be used to accurately predict the learning curves of DNNs.

Fitness Prediction with Sequence-to-Sequence Network. Another approach similar to surrogate optimization for reducing computation time is fitness prediction with sequence-to-sequence models. Such models are powerful approaches that make use of recurrent neural networks to learning a mapping between two different sequences of multi-dimensional vectors [143]. As shown in Figure 7.2, the network is composed of two LSTM layers, one called the encoder and another called the decoder. A sequence of arbitrary length is given as input to the encoder and the encoder is used to generate an embedding from its hidden-state representation. This embedding is then passed to the decoder (which shares the same architecture and number of parameters) as the encoder and another sequence is outputted. The model can be trained so that when given an input sequence it can predict a particular output sequence that is associated or related with the input.

Sequence-to-sequence models can be used to improve the performance of CoDeepNEAT by reducing the number of epochs require for training during evaluation of the assembled networks. For example, some of the assembled networks are trained for a reduced number of epochs while the remaining are fully trained. The data points (such as training loss and fitness after every epoch) can be used to train a model to predict the full training curves of networks that are only partially trained. Such method can significantly reduce the computational budget required to perform evolutionary architecture search and the number of generations required for fitness to converge.

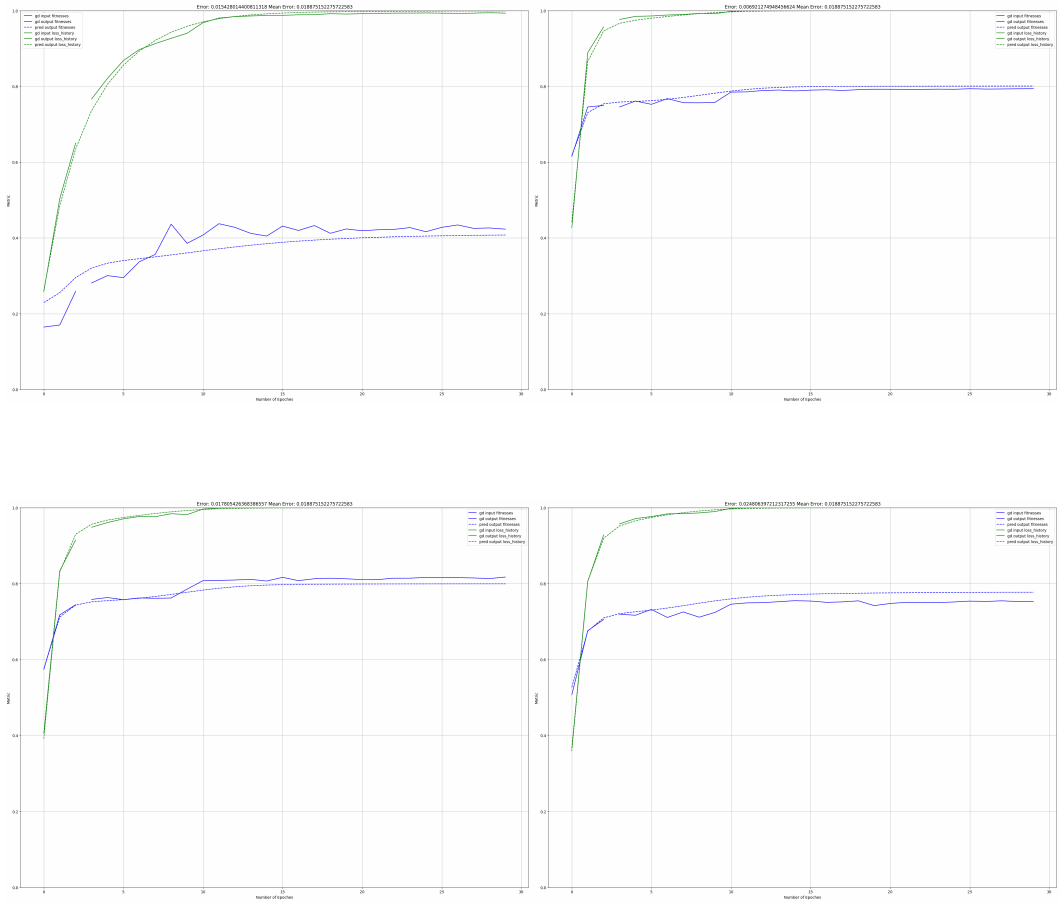


Figure 7.3: Visualizations of the example predictions of a sequence-to-sequence model when given three epochs of training loss (green) and fitness (blue) as input. The solid line shows the ground-truth values while the break shows the first few epochs that were given as input to the model. The dotted line shows the predicted values of the model. The model is accurate and able to predict within 2% error of the ground-truth.

Preliminary experiments in applying a sequence-to-sequence model to predict the training curves of networks in the Omniglot domain are promising.

The model is trained on training data (both training loss and fitness on validation set) from approximately 300 networks and then tested on a holdout set of 100 networks. The input sequence length was only three and much smaller when compared to the length of the full sequence (30). However, as Figure 7.3 shows, the predicted training curves (dotted lines) are very close to the actual training curves (solid lines) for both the loss (green) and the fitness (blue). The average error is with 2% and shows the power of the model even when given very little training data. Future work will attempt to perform online learning of the model during evolution to substantially reduce the training times of many networks.

Hyperparameter Fine-tuning with CMA-ES. CMA-ES is a state-of-the-art EA for the optimization of fixed-length real-valued vectors [47, 91]. The algorithm works well because it can capture interactions between dimensions. While CMA-ES can outperform other EAs, it cannot optimize the architecture of DNNs by itself. However, it is conceivable that after the evolution of a DNN's architecture through CodeepNEAT, any real-valued hyperparameters can continue to be optimized using CMA-ES. These values include critical training parameters such as learning rate and momentum, weight initialization, weight regularization, and various data augmentation parameters.

Preservation of Network Weights During Mutation and Evaluation. One of the main sources of computational complexity in the current framework for training and evaluating DNNs is that the networks have to be trained from scratch. Even if a network remains unchanged from one generation to the

next, any previous effort spent on training it during the previous generation is completely wasted. The number of epochs that each network can be trained is limited to a relatively low number, leading to premature fitness convergence in CoDeepNEAT and creates an undesirable bias towards networks that learn very fast, but also overfit when trained for longer.

This problem can be solved by sending back the weights/parameters of the trained networks along with the fitness from the workers, storing the weights, and sending the weights along with the DNN for evaluation during the next generation. Consequently, a network with pretrained weights will not start training from scratch and if it stays in the population long enough, the network might be trained until its loss converges. However, there are still two issues with this solution: (1) Networks that undergo mutations that add new layers or connections between layers cannot simply use the weights of the parent. In this case, something similar to Net2Net [21] can be implemented, where new weights parameters are added in a way that preserves the behavior of the network. (2) Weight preservation is biased towards older, well trained networks and might lead to new but promising networks being crowded out. This bias can be alleviated by using a regression model that incorporates features describing the network's actual performance to predict its converged fitness; this predicted fitness is then used in place of the original fitness.

All of the proposed future work for CoDeepNEAT require minimal changes to the core algorithm and have the potential to significantly improve performance or reduce computation time.

7.2 CoDeepNEAT-AES

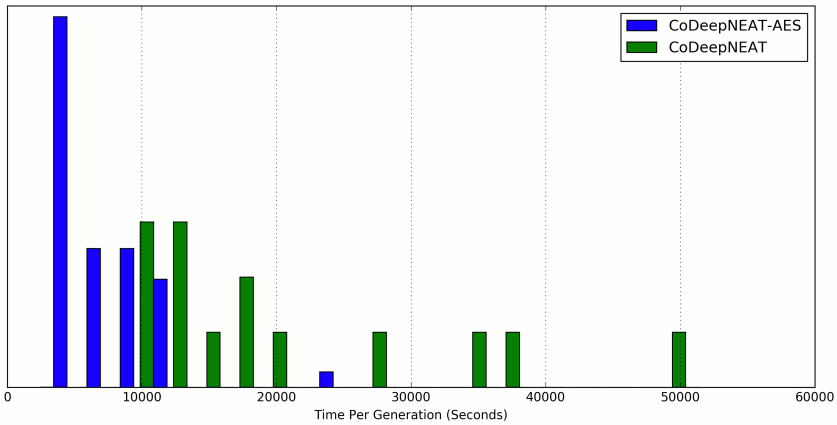


Figure 7.4: Histogram of time per generation for synchronous CoDeepNEAT and CoDeepNEAT-AES. The average time per generation for CoDeepNEAT-AES is significantly less than that for CoDeepNEAT.

As the experimental results show, the asynchronous CoDeepNEAT-AES provides significant speedups when compared to its synchronous baseline. Furthermore, the hyperparameter D, which controls the ratio between K (initial population size) and M (number of results to wait for), has a massive impact on the performance for CoDeepNEAT-AES. In the case where $D = 1$ ($M = K$), CoDeepNEAT-AES becomes identical to a synchronous evaluation strategy and slow for the reasons mentioned in Section 4.3. Interestingly enough, setting a value for D that is too large also hurts performance because as M becomes smaller, both the returned individuals and the new population that is generated from them become less diverse.

The histogram in Figure 7.5 reveals how CoDeepNEAT-AES improves

performance over a synchronous evaluation strategy. This plot visualizes the relative frequency at which individuals (along with their fitness) return from the completion service over the duration of a typical generation. In the synchronous version of of CoDeepNEAT, individuals in the population are submitted and all come back in the same generation before evolution can proceed. As a result the histogram for synchronous CoDeepNEAT resembles a Gaussian distribution with only a few individuals returning early and later. Thus, much time is spent waiting for the last few individuals to return at the end of a generation. On the other hand, less time is wasted with CoDeepNEAT-AES, as indicated by the flat distribution in the histogram. Individuals are returned at a very steady, regular rate over the course of a generation and there are no slow individuals that might bottleneck the EA.

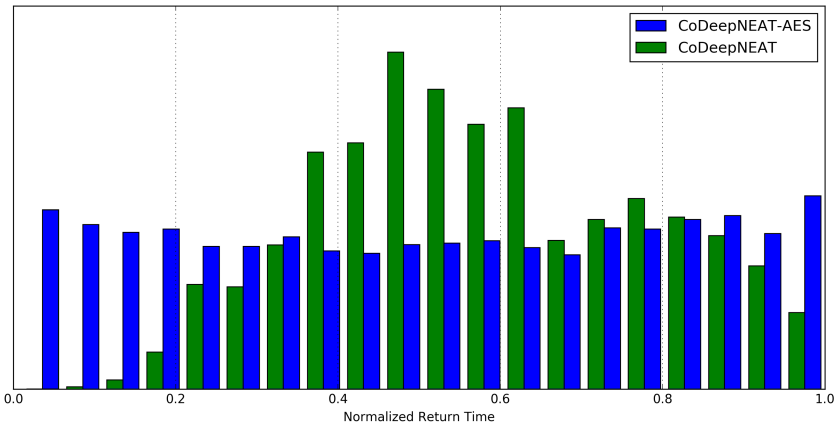


Figure 7.5: Histogram of frequency of returned results over the course of a typical generation for both algorithms. CoDeepNEAT-AES wastes less time because the result results have a flat distribution compared to the Gaussian distribution for CoDeepNEAT.

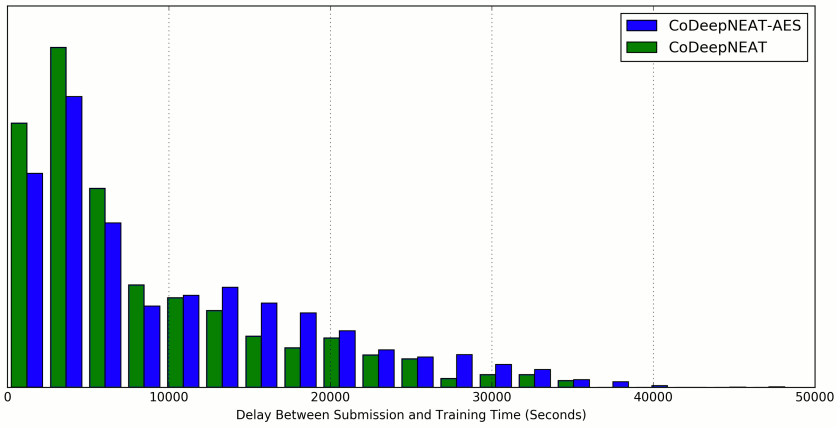


Figure 7.6: Histogram comparing the delay between submission of individuals by the EA and when they are actually trained. The average delay time is longer for CoDeepNEAT-AES, but does not seem to negatively affect performance.

There is one statistic where the synchronous version of CoDeepNEAT has an advantage and can be seen in the histogram in Figure 7.6. This histogram visualizes the time delay between when an individual is submitted by the server to the completion service and when that same individual is evaluated (trained) by a worker node. The delay amount is slightly higher on average for CoDeepNEAT-AES and is probably because CoDeepNEAT-AES maintains more individuals on average on the completion service submission buffer. However, as the fitness plot in Figure 4.10 indicate, having a higher delay does not negatively affect performance.

In the future, it will be interesting to explore how the distribution of evaluation times for the population affects the speed-up provided by asynchrony. Second, more extensive experiments can be done to analyze how dif-

ferent values for K and D will affect the performance of CoDeepNEAT-AES. Third, more work can be done to determine how CoDeepNEAT-AES will perform in a heterogeneous distributed environment where the compute resources can vary in terms of performance and hardware configuration.

Asynchronous evaluations are a promising method to speed up evolutionary architecture search and as shown by CoDeepNEAT-AES, can be easily integrated into a synchronous EA.

7.3 CM and CMSR

The experiments show that MTL can improve performance significantly across tasks, and that the architecture used for it matters a lot. Multiple ways of optimizing the architecture are proposed in this paper and the results lead to several insights.

First, modules used in the architecture can be optimized and they do end up different in a systematic way. Unlike in the original soft-ordering architecture, evolution in CM and CMSR discovers a wide variety of simple and complex modules, and they are often repeated in the architecture. They thus serve as building blocks that are diverse in structure.

Second, the routing of the modules matter as well. In CMSR, the shared but evolvable routing allows much more flexibility in how the modules can be reused, extending the principles that make soft ordering useful. If indeed the power from multitasking comes from integrating requirements of multiple

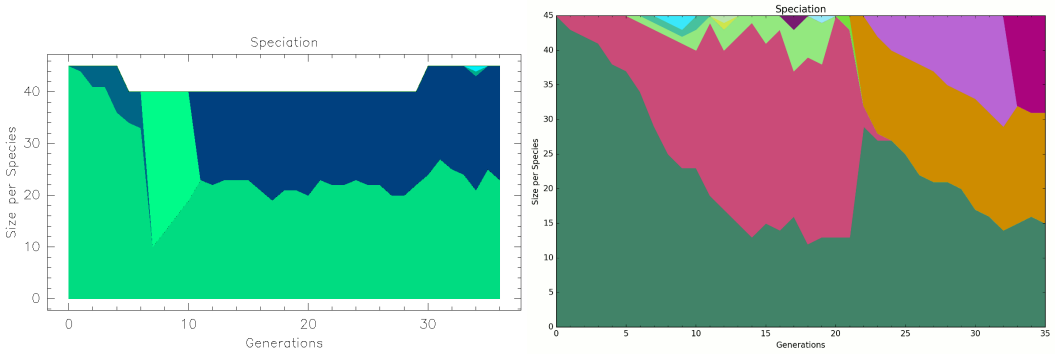
tasks, this integration will happen in the embeddings that the modules form, so it makes sense that sharing plays a central role.

Third, sharing components (including weight values) in CM can result in powerful networks that are better than the original soft-ordering architecture, However, CMSR often evolves away from sharing module weights, despite the fact that module architectures are often reused in the network. This result makes sense as well: Because the topology is shared in this approach, the differentiation between tasks comes from differentiated modules. While shared topology is effective on its own, experimental results in Table 5.1 show that data augmentation leads to even better performance. These results suggest that the performance boost that CMSR gives is orthogonal to other methods to improve network generalization and can be combined with techniques such as more advanced data augmentation, ensembling, and cyclical learning rate scheduling.

There are several directions for future work in CMSR in particular. The proposed algorithm can be extended to many applications that lend themselves to the multitask approach. For instance, it will be interesting to see how it can be used to find synergies in different tasks in vision and language. Furthermore, as was shown in related work, the tasks do not even have to be closely related to gain the benefit from MTL. For instance, object recognition can be paired with caption generation. It is possible that the need to express the contents of an image in words will help object recognition, and vice versa. Discovering ways to tie such multimodal tasks together should be a good opportunity

for evolutionary optimization, and constitutes a most interesting direction for future work.

7.4 MCDN and MCMSR



(a) CoDeepNEAT (b) MCDN

Figure 7.7: Visualization of the number of unique species created for both CoDeepNEAT (Figure 7.7a) and MCDN (Figure 7.7b) during evolution. Each species are represented as a unique color, with the X-axis showing the current generation. The increased number of species created by MCDN shows that it is able to maintain a more diverse population through using novelty as a secondary objective.

The faster convergence of MCDN compared to CoDeepNEAT shows that novelty indeed helps to overcome deception and local optima in the architecture search space. While both MCDN and CoDeepNEAT start from minimal topologies, MCDN complexifies network structure faster and favors individuals whose structure is more unique than the rest of the population. As a result, it maintains a more diverse population pool which is critical in

preventing premature convergence [83]. The increased diversity of the the population pool can be visualized through speciation mechanism in CoDeepNEAT. As shown in Figure 7.7, the number of unique species created during evolution in the module population pool is higher in MCDN than in CoDeepNEAT. In CoDeepNEAT, species rarely die out, but in MCDN, new species are created and become extinct at a higher rate.

The experiment with MCMSR shows that multiobjective optimization is effective in discovering networks that can balance trade-offs between multiple metrics. As seen in the Pareto fronts of Figure 6.6, the networks discovered by MCMSR dominate those evolved by CoDeepNEAT with respect to the complexity and fitness metrics in almost every generation. Interestingly enough, this domination is not symmetrical and evolves over time. In generation 10, networks discovered by MCMSR and CoDeepNEAT have similar average complexity but those evolved by MCMSR have much higher average fitness. This situation changes later in evolution; by generation 40, the average complexity of the networks discovered by MCMSR is noticeably lower than that of CoDeepNEAT, but the gap in average fitness between MCMSR and CoDeepNEAT has also narrowed. In other words, MCMSR first tries optimize for the first objective (fitness)and only when fitness is starting to converge, does it try to improve the second objective (network complexity). Thus, multiobjective evolution favors progress in the metric that is easiest to improve upon at the moment and moves in the direction of least resistance. MCMSR does not get stuck on one objective; it will try to optimize another objective if no progress

is made on the current one.

More surprisingly, MCMSR maintains a higher average fitness with each generation (Figure 6.5). This finding suggests that minimizing network complexity produces a regularization effect that also improves the generalization of the network. This effect could be due to the fact that networks evolved by MCMSR reuse modules more often when compared to CoDeepNEAT; extensive module reuse has been shown to improve performance in many hand-designed architectures [51, 145].

7.5 Conclusion

In this chapter, detailed analysis is carried out for the evolutionary architecture search algorithms introduced in this dissertation. The successes and limitations of these algorithms are explored and several interesting directions of future work are proposed. In particular, the two most promising areas are surrogate optimization and fitness prediction from partial training. Preliminary experimental results suggest that these two improvements might have a large impact on the performance of evolutionary architecture search.

Chapter 8

Conclusion

One of the biggest challenges faced in deep learning research today is determining a suitable architecture and hyperparameters for deep neural network (DNN) models. The success of deep learning in many problem domains depends on solving this problem. For example, recent innovations and advances in the large-scale image classification domain are mostly due to innovations in the design of DNN architecture [145]. However, as deep learning has scaled up to more challenging tasks, the architectures have become difficult to design by hand. Subtle changes in network architecture can have a large impact on performance, and as a result, it is difficult to tell what the right architecture is for a given problem. In many deep models, the choice of architecture/hyperparameters are often made based on history and convenience, without extensive experimentation and testing. Such an informal architecture optimization is most done by hand or by inefficient methods such as grid search [149]. Unfortunately, these methods do not scale with the increasingly complex architectures and hyperparameters often found in state-of-the-art networks.

Since significant gains in performance could be extracted with the right

DNN configuration, there has been considerable research into automated methods for optimizing DNN hyperparameters such as Bayesian optimization [137]. Such methods usually assume a fixed architecture and optimize the hyperparameters only. While reinforcement learning based methods for architecture search exist [169], they are restricted to optimizing within a limited search space. This dissertation shows that an evolutionary approach to designing network architectures can scale up to finding the right network architectures in previously intractable search spaces. Given the anticipated increases in available computing power, evolution of deep networks is promising approach to constructing deep learning applications in the future. The following section will list the significant contributions of this dissertation.

8.1 Contributions

Extending Neuroevolution to Deep Learning. This dissertation made a connection between the vast amount of previous work done in evolving neural networks with evolving DNN topology. By extending existing neuroevolution methods to topology, components, and hyperparameters, an efficient EA for architecture search called DeepNEAT was created. Powerful heuristics such as incrementally complexifying DNNs and protecting innovation through specification allow DeepNEAT to evolve networks with arbitrary graph structure and hyperparameters.

Extending Coevolution to Neural Architecture Search. Coevolution is another innovation in DNN architecture search that is inspired by evolution-

ary computation. CoDeepNEAT utilizes the principles behind coevolution by evolving modules and blueprints; they are then combined to form an assembled network for evaluation. CoDeepNEAT discovers promising architectures within a much larger search space, including modular, repetitive structures commonly seen in state-of-the-art DNNs.

Improving Performance through Asynchronous Evolution. CoDeepNEAT can be parallelized by distributing network evaluations to a cluster of machines, but the drawback of such an approach is that many machines will be idle at the end of a generation. An asynchronous version of CoDeepNEAT is proposed that minimizes idle time and enables CoDeepNEAT to converge significantly faster given a fixed computational budget. As a result, evolutionary approaches to architecture search become less computationally complex.

Adapting Evolutionary Architecture Search to Multitask Learning. This dissertation introduced a novel version of CoDeepNEAT called CMSR that is specially designed for evolving deep multitask networks. CMSR extends previous work that showed carefully designed routing and sharing of modules can help multitask learning significantly. By evolving modules that can be used in different ways by different tasks and incorporating a recent innovation called soft-ordering, each task is processed differently in the evolved network. As a result, CMSR achieves significantly better results than conventional neural architectures in the multitask learning domain.

Evolutionary Multiobjective Architecture Search. Another major contribution is the application of multiobjective optimization to neural architecture

search. This application is done through a novel, multiobjective variant of CoDeepNEAT that searches for a Pareto front of solutions, trading off two or more objectives. This mechanism allows discovery of networks that not only perform well but also have specific properties such as small size.

Competitive Results in Multiple Problem Domains. The methods proposed in this dissertation achieve results comparable or exceeding the best human designs in standard benchmarks in image classification, image captioning, and multitask learning. This is especially true in the Omniglot domain, where the evolved network beats the previous state-of-the-art by a large margin of 22%. Just as important is the fact the networks evolved by CoDeepNEAT and its variants are often smaller in size and faster to train than their hand-designed counterparts.

Real World Applications of Evolutionary Architecture Search. The dissertation also presented a case-study where a evolved network was used to build a real-world application of automated image captioning on a magazine website. The case-study shows the commercial viability of CoDeepNEAT and its wide applicability to solve difficult and challenging problems commonly seen in the real world.

8.2 Concluding Remarks

The work in this dissertation addressed the need and opportunity for evolutionary computation in designing the next generation of deep learning systems. The topology, components, and hyperparameters of the architecture

can all be optimized simultaneously to fit the requirements of the task, resulting in superior performance. Currently such designs are comparable or slightly better than the best human designs; with steady increase in available computing power, they should soon surpass them by a large margin, putting the power to good use. This dissertation already showed that many different domains can benefit from evolutionary neural architecture search; many more areas remain where this technology can be applied to improve the user experience or improve productivity. Such automated design can make new and unexpected applications of deep learning possible in vision, speech, language, and other areas. In the long term, hand-design of algorithms and DNNs may be fully replaced by more sophisticated and general-purpose automated systems to aid scientists in their research or engineers in designing an AI-enabled product.

Bibliography

[1] Amazon web services (aws) - cloud computing services. aws.amazon.com.

[2] Jigsaw toxic comment classification challenge. kaggle.com/jigsaw-toxic-comment-classification-challenge.

[3] Mightyai vision products. mty.ai/computer-vision.

[4] Moe. github.com/Yelp/MOE.

[5] Multiobjective optimization. en.wikipedia.org/wiki/Multi-objective_optimization.

[6] Research:detox/data release. meta.wikimedia.org/wiki/Research:Detox.

[7] Sentient technologies. sentient.ai.

[8] Seq2seq. github.com/farizrahman4u/seq2seq.

[9] Studioml. https://www.studio.ml/.

[10] Using the microsoft tlc machine learning tool. jamesmccaffrey.wordpress.com/2017/01/13/using-the-microsoft-tlc-machine-learning-tool.

[11] Wikipedia. en.wikipedia.org/wiki/Main_Page.

[12] Automl for large scale image classification and object detection. https://ai.googleblog.com/2017/11/automl-for-large-scale-image.html, Nov 2017.

[13] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems 19*, 2007.

[14] A. Argyriou, T. Evgeniou, and M. Pontil. Convex multi-task feature learning. *Machine Learning*, 73(3):243–272, Dec 2008.

[15] James Bagnell and Jeff Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the International Conference on Robotics and Automation 2001*. IEEE, May 2001.

[16] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[17] H. Bilen and A. Vedaldi. Universal representations: The missing link between faces, text, planktons, and cat breeds. *CoRR*, abs/1701.07275, 2017.

[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[19] R. Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer US, 1998.

[20] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. *CoRR*, abs/1606.01865, 2016.

[21] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[22] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*, 2015.

[23] Theodora Chu, Kylie Jue, and Max Wang. Comment abuse classification with deep learning. *Von https://web. stanford. edu/class/cs224n/reports/2762092. pdf abgerufen*, 2016.

[24] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, pages 160–167, 2008.

[25] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[26] Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007.

[27] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[28] Coline D., Abhishek G., Trevor D., et al. Learning modular neural network policies for multi-task and multi-robot transfer. *CoRR*, abs/1609.07088, 2016.

[29] Kalyanmoy Deb. Multi-objective evolutionary algorithms. In *Springer Handbook of Computational Intelligence*, pages 995–1015. Springer, 2015.

[30] Kalyanmoy Deb and Christie Myburgh. Breaking the billion-variable barrier in real-world optimization using a customized evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 653–660. ACM, 2016.

[31] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[32] Matjaz Depolli, Roman Trobec, and Bogdan Filipic. Asynchronous master-slave parallelization of differential evolution for multi-objective optimization. *Evolutionary computation*, 21:261–291, 213.

[33] D. Dong, H. Wu, W. He, et al. Multi-task learning for multiple language translation. In *ACL*, pages 1723–1732, 2015.

[34] Thomas Elsken, J Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *ArXiv e-prints*, 1804.

[35] Thomas Elsken, J Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *ArXiv e-prints*, 2018.

[36] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[37] T. Evgeniou and M. Pontil. Regularized multi–task learning. In *Proc. of KDD*, pages 109–117, 2004.

[38] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: From architectures to learning. *Evolutionary Intelligence*, 1:47–62, 2008.

[39] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[40] Ben Goertzel and Cassio Pennachin. *Artificial general intelligence*, volume 2. Springer, 2007.

[41] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

[42] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.

[43] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.

[44] Faustino J Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, pages 1356–1361, 1999.

[45] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE, 2013.

[46] Frederic Gruau and Darrell Whitley. Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. *Evolutionary Computation*, 1:213–233, 1993.

[47] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with co-

variance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[48] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Hypercolumns for object segmentation and fine-grained localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 447–456, 2015.

[49] K. Hashimoto, C. Xiong, Y. Tsuruoka, and R. Socher. A joint many-task model: Growing a neural network for multiple NLP tasks. *CoRR*, abs/1611.01587, 2016.

[50] Matthew Hausknecht, Piyush Khandelwal, Risto Miikkulainen, and Peter Stone. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 217–224. ACM, 2012.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.

[53] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[54] Geoffrey E. Hinton and Steven J. Nowlan. How learning can guide evolution. *Complex Systems*, 1:495–502, 1987.

[55] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

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

[57] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

[58] J. T. Huang, J. Li, D. Yu, et al. Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In *Proc. of ICASSP*, pages 7304–7308, 2013.

[59] Z. Huang, J. Li, S. M. Siniscalchi, et al. Rapid adaptation for deep neural networks through multi-task learning. In *Proc. of INTERSPEECH*, 2015.

[60] J. Huizinga, J.-B. Mouret, and J. Clune. Does aligning phenotypic and genotypic modularity improve the evolution of neural networks? In *Proc. of GECCO*, pages 125–132, 2016.

[61] Christian Igel, Thorsten Suttorp, and Nikolaus Hansen. Steady-state selection and efficient covariance matrix update in the multi-objective cma-es. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 171–185. Springer, 2007.

[62] M. Jaderberg, V. Mnih, W. M. Czarnecki, et al. Reinforcement learning with unsupervised auxiliary tasks. In *ICLR*, 2017.

[63] W. Jaśkowski, K. Krawiec, and B. Wieloch. Multitask visual learning using genetic programming. *Evolutionary Computation*, 16(4):439–459, 2008.

[64] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[65] L. Kaiser, A. N. Gomez, N. Shazeer, et al. One model to learn them all. *CoRR*, abs/1706.05137, 2017.

[66] Z. Kang, K. Grauman, and F. Sha. Learning with whom to share in multi-task feature learning. In *Proc. of ICML*, pages 521–528, 2011.

[67] S. Kelly and M. I. Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *Proc. of GECCO*, pages 195–202, 2017.

[68] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In *Encyclopedia of machine learning*, pages 257–258. Springer, 2011.

[69] Jinwoo Kim. *Hierarchical asynchronous genetic algorithms for parallel/distributed simulation-based optimization*. PhD thesis, University of Arizona, 1994.

[70] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[71] G. Koch, R. Zemel, and R. Salakhutdinov. Siamese neural networks for one-shot image recognition. In *Proc. of ICML*, 2015.

[72] Rogier Koppejan and Shimon Whiteson. Neuroevolutionary reinforcement learning for generalized control of simulated helicopters. *Evolutionary Intelligence*, 4:219–241, 2011.

[73] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[74] A. Kumar and H. Daumé, III. Learning task grouping and overlap in multi-task learning. In *Proc. of ICML*, pages 1723–1730, 2012.

[75] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[76] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[77] John Boaz Lee and Xiangnan Kong. Skip-graph: Learning graph embeddings with an encoder-decoder model. 2016.

[78] Joel Lehman and Risto Miikkulainen. Neuroevolution. *Scholarpedia*, 8(6):30977, 2013.

[79] Joel Lehman and Risto Miikkulainen. Neuroevolution. *Scholarpedia*, 8(6):30977, 2013.

[80] Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[81] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[82] Joel Lehman and Kenneth O Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218. ACM, 2011.

[83] Joel Lehman, Kenneth O Stanley, and Risto Miikkulainen. Effective diversity maintenance in deceptive domains. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 215–222. ACM, 2013.

[84] Jason Liang, Elliot Meyerson, and Risto Miikkulainen. Evolutionary architecture search for deep multitask networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 466–473, New York, NY, USA, 2018. ACM.

[85] Jason Zhi Liang and Risto Miikkulainen. Evolutionary bilevel optimization for complex control tasks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2015)*, Madrid, Spain, July 2015.

[86] Jason Zhi Liang and Risto Miikkulainen. Evolutionary bilevel optimization for complex control tasks. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 871–878. ACM, 2015.

[87] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[88] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017.

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

[90] X. Liu, J. Gao, X. He, L. Deng, K. Duh, and Y. Y. Wang. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. In *Proc. of NAACL*, pages 912–921, 2015.

[91] Ilya Loshchilov and Frank Hutter. Cma-es for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, 2016.

[92] Y. Lu, A. Kumar, S. Zhai, et al. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. *CVPR*, 2017.

[93] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: A multi-objective genetic algorithm for neural architecture search. *arXiv preprint arXiv:1810.03522*, 2018.

[94] Sean Luke. *The ECJ Owners Manual*, 22nd edition, August 2014.

[95] M.-T. Luong, Q. V. Le, I. Sutskever, et al. Multi-task sequence to sequence learning. In *Proc. of ICLR*, 2016.

[96] D. Maclaurin, D. Duvenaud, and R. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proc. of ICML*, pages 2113–2122, 2015.

[97] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[98] Georgios Methenitis, Daniel Hennes, Dario Izzo, and Arnoud Visser. Novelty search for soft robotic space exploration. In *Proceedings of the 2015 annual conference on Genetic and Evolutionary Computation*, pages 193–200. ACM, 2015.

[99] E. Meyerson and R. Miikkulainen. Beyond shared hierarchies: Deep multitask learning through soft layer ordering. *ICLR*, 2018.

[100] Tomáš Mikolov. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 2012.

[101] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[102] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert. Cross-stitch networks for multi-task learning. In *Proc. of CVPR*, 2016.

[103] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[104] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 762–767. San Francisco: Morgan Kaufmann, 1989.

[105] Jacqueline Moore, Richard Chapman, and Gerry Dozier. Multiobjective particle swarm optimization. In *Proceedings of the 38th annual on Southeast regional conference*, pages 56–57. ACM, 2000.

[106] David E. Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive co-evolution. *Evolutionary Computation*, 5:373–399, 1997.

[107] David E Moriarty and Risto Miikkulainen. Hierarchical evolution of neural networks. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 428–433. IEEE, 1998.

[108] Andrew Y. Ng, H. Jin Kim, Michael Jordan, and Shankar Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems 16*, 2004.

[109] Joe Yue-Hei Ng, Matthew J. Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. *CoRR*, abs/1503.08909, 2015.

[110] Una-May OReilly, Mark Wagy, and Babak Hodjat. Ec-star: A massive-scale, hub and spoke, distributed genetic programming system. In *Genetic Programming Theory and Practice X*, pages 73–85. Springer, 2013.

[111] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

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

[113] Mitchell A Potter and Kenneth A De Jong. A cooperative coevolutionary approach to function optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 249–257. Springer, 1994.

[114] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225*, 2017.

[115] R. Ranjan, V. M. Patel, and R. Chellappa. Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition. *CoRR*, abs/1603.01249, 2016.

[116] Khaled Rasheed and Brian D. Davison. Effect of global parallelism on the behavior of a steady state genetic algorithm for design optimization. In *In Proceedings of the 1999 Congress on Evolutionary Computation,*. IEEE, 1999.

[117] Aditya Rawal and Risto Miikkulainen. From nodes to networks: Evolving recurrent neural networks. *arXiv preprint arXiv:1803.04439*, 2018.

[118] E. Real, S. Moore, A. Selle, et al. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[119] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

[120] S.-A. Rebuffi, H. Bilen, and A. Vedaldi. Learning multiple visual domains with residual adapters. In *NIPS*, pages 506–516. 2017.

[121] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[122] D. Rezende, M. Shakir, I. Danihelka, K. Gregor, and D. Wierstra. One-shot generalization in deep generative models. In *ICML*, pages 1521–1529, 2016.

[123] S. Ruder. An overview of multi-task learning in deep neural networks. *CoRR*, abs/1706.05098, 2017.

[124] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[125] J. Schrum and R. Miikkulainen. Solving multiple isolated, interleaved, and blended tasks through modular neuroevolution. *Evolutionary Computation*, 24(3):459–490, 2016.

[126] Jacob Schrum, Igor V Karpov, and Risto Miikkulainen. Ut 2: Human-like behavior via neuroevolution of combat behavior and replay of human traces. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 329–336. IEEE, 2011.

[127] Jacob Schrum and Risto Miikkulainen. Constructing complex npc behavior via multi-objective neuroevolution. *AIIDE*, 8:108–113, 2008.

[128] Eric O. Scott and Kenneth A. De Jong. Understanding simple asynchronous evolutionary algorithms. In *Foundations of Genetic Algorithms*. 2015.

[129] M. L. Seltzer and J. Droppo. Multi-task learning in deep neural networks for improved phoneme recognition. In *Proc. of ICASSP*, pages 6965–6969, 2013.

[130] Hormoz Shahrzad, Daniel Fink, and Risto Miikkulainen. Enhanced optimization with composite objectives and novelty selection. *arXiv preprint arXiv:1803.03744*, 2018.

[131] Hormoz Shahrzad and Babak Hodjat. Tackling the boolean multiplexer function using a highly distributed genetic programming system. In *Genetic Programming Theory and Practice XII*, pages 167–179. Springer, 2015.

[132] P. Shyam, S. Gupta, and A. Dukkipati. Attentive recurrent comparators. In *Proc. of ICML*, pages 3173–3181, 2017.

[133] Ankur Sinha, Pekka Malo, Peng Xu, and Kalyanmoy Deb. A bilevel optimization approach to automated parameter tuning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014)*, Vancouver, BC, Canada, July 2014.

[134] Leslie N Smith. Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pages 464–472. IEEE, 2017.

[135] M. Snel and S. Whiteson. Multi-task evolutionary shaping without pre-specified representations. In *GECCO*, pages 1031–1038, 2010.

[136] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[137] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Mr Prabhat, and Ryan P Adams. Scalable bayesian optimization using deep neural networks. In *ICML*, pages 2171–2180, 2015.

[138] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005.

[139] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[140] Kenneth O Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 569–577. Morgan Kaufmann Publishers Inc., 2002.

[141] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10:99–127, 2002.

[142] M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proc. of GECCO*, pages 497–504. ACM, 2017.

[143] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[144] C. Szegedy, V. Vanhoucke, S. Ioffe, et al. Rethinking the inception architecture for computer vision. In *Proc. of CVPR*, pages 2818–2826, 2016.

[145] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[146] Y. Teh, V. Bapst, W. M. Czarnecki, et al. Distral: Robust multitask reinforcement learning. In *NIPS*, pages 4499–4509. 2017.

[147] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[148] Fjodor van Veen. The neural network zoo. `asimovinstitute.org/neural-network-zoo`, Oct 2016.

[149] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning

useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(Dec):3371–3408, 2010.

[150] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

[151] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE transactions on pattern analysis and machine intelligence*, 39(4):652–663, 2017.

[152] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 3462–3471. IEEE, 2017.

[153] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[154] Shimon Whiteson and Peter Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7(May):877–917, 2006.

[155] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[156] Brian G Woolley and Kenneth O Stanley. Exploring promising stepping stones by combining novelty search with interactive evolution. *arXiv preprint arXiv:1207.6682*, 2012.

[157] Z. Wu, C. Valentini-Botinhao, O. Watts, and S. King. Deep neural networks employing multi-task learning and stacked bottleneck features for speech synthesis. In *Proc. of ICASSP*, pages 4460–4464, 2015.

[158] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[159] Feng Xue, Arthur C Sanderson, and Robert J Graves. Pareto-based multi-objective differential evolution. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 2, pages 862–869. IEEE, 2003.

[160] Kohsuke Yanai and Hitoshi Iba. Multi-agent robot learning by means of genetic programming: Solving an escape problem. In *International Conference on Evolvable Systems*, pages 192–203. Springer, 2001.

[161] Y. Yang and T. Hospedales. Deep multi-task representation learning: A tensor factorisation approach. In *Proc. of ICLR*, 2017.

[162] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[163] Chern Han Yong and Risto Miikkulainen. Cooperative coevolution of multi-agent systems. *University of Texas at Austin, Austin, TX*, 2001.

[164] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image captioning with semantic attention. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4651–4659, 2016.

[165] Bernard P Zeigler. Asynchronous genetic algorithms on parallel computers. In *Proc. International Conference on Genetic Algorithms, June, 1993*, volume 660, 1993.

[166] Y. Zhang and D. Weiss. Stack-propagation: Improved representation learning for syntax. *CoRR*, abs/1603.06598, 2016.

[167] Z. Zhang, L. Ping, L. C. Chen, and T. Xiaoou. Facial landmark detection by deep multi-task learning. In *Proc. of ECCV*, pages 94–108, 2014.

[168] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. Multiobjective evolutionary

algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32–49, 2011.

[169] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

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

Vita

Jason Zhi Liang received his Bachelors degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 2013. As an undergraduate there, he worked with Avideh Zakhor on Computer Vision algorithms for image retrieval and localization in indoor environments. As a masters student at the University of Texas at Austin, he worked with Risto Miikkulanien on bilevel evolutionary algorithms. Currently, as a PhD student, he is researching evolutionary methods for evolving deep neural network architectures. He is currently pursing a doctorate degree in Computer Science under the supervision of Risto Miikkulainen at the University of Texas at Austin and will graduate in 2018.

Permanent address: jasonzliang@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.