CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Evolving Deep Neural Networks: Optimization of Weights and Architectures

Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) vorgelegt von

Jonas Prellberg

Tag der Disputation
08.07.2020

Gutachter
Prof. Dr. Oliver Kramer
Jun.-Prof. Dr. Paul Kaufmann

Zusammenfassung

Deep neural networks sind flexible statistische Modelle, welche eine Vielzahl von Funktionen darstellen können. Im Kontext von überwachtem Lernen ist es möglich, die Netze anhand von Daten zu trainieren, so dass sie die zugrunde liegende Funktion zwischen Eingaben und Ausgaben annähern. In vielen Fällen generalisieren die Netze gut für unbekannte Eingaben, womit sie zu mächtigen Vorhersagemodellen werden, die in verschiedenen Disziplinen sehr erfolgreich angewendet werden. Jedoch ist es eine schwierige Aufgabe ein gut generalisierendes Modell zu trainieren, da viel Rechenleistung, große Datensätze und die korrekte Wahl einer großen Anzahl an Hyperparametern erforderlich sind.

Da es keine theoretische Grundlage für den Einfluss vieler Hyperparameter gibt, können gute Einstellungen nur durch Experimente gefunden werden. Auf Grund des großen Hyperparametersuchraums und dem rechenaufwändigen Training kann nur ein winziger Teil des Suchraums manuell untersucht werden. Konsequenterweise werden Hyperparameter häufig durch Erfahrungswerte und eine geringe Anzahl an Experimenten bestimmt. In dieser Dissertation werden wir die Rolle der Netzwerkarchitektur als besonders wichtigen Hyperparameter herausstellen und automatische Methoden zu ihrer Optimierung entwickeln. Die Gewichte und Architektur neuronaler Netze sind eng miteinander verbunden und es existieren verschiedene Ansätze, um beide zu optimieren.

Zuerst werden wir Ideen aus dem Forschungsgebiet der Neuroevolution untersuchen und sie auf *deep neural networks* hochskalieren. Dies wird ermöglicht durch eine GPU-beschleunigte Implementierung eines evolutionären Algorithmus für das Training neuronaler Netze. Wir untersuchen seine Leistungsfähigkeit auf überwachten Lernproblemen und ziehen Vergleiche zu stochastischen Gradientenabstiegsverfahren. Ein ähnlicher evolutionärer Algorithmus für das Training neuronaler Netze wird im Kontext von bestärkendem Lernen demonstriert. Beide Ansätze zeigen die Fähigkeit evolutionärer Algorithmen auf, große neuronale Netze zu trainieren, aber der Algorithmus zum überwachten Lernen ist ineffizient, da er den zur Verfügung stehenden Gradienten ignoriert.

Weil wir unseren Fokus auf das überwachte Lernen setzen, wenden wir uns dem Forschungsgebiet der neuronalen Architektursuche zu, in welchem Gradientenabstiegsverfahren für das Gewichtstraining verwendet werden. Unser erster evolutionärer *black-box* Architektursuchalgorithmus verwendet eine Gewichtsvererbungsstrategie, die die benötigte Rechenzeit für den Suchprozess signifikant verringert. Das ist eine

wichtige Errungenschaft, da neuronale Architektursuche ungeheure Mengen an Rechenzeit benötigen kann. In unserem zweiten neuronalen Architektursuchalgorithmus entfernen wir die *black-box*-Annahme und verringern die Rechenzeit weiter, indem Gewichte und Architektur zusammen in einem einzigen Durchlauf optimiert werden. In dem Bemühen die Anzahl der benötigten Trainingsbeispiele zu verringern, nutzen wir unseren Algorithmus um *multi-task* Lernen durchzuführen und zeigen gute Ergebnisse im Vergleich mit anderen *multi-task* Lernalgorithmen.

Abstract

Deep neural networks are flexible statistical models that can represent a large number of different functions. In a supervised learning setting, they can be trained from data to approximate the underlying function between inputs and outputs. In many cases, they generalize well to unseen inputs, which makes them powerful predictive models that are applied in many different fields of study to great success. However, training a model that generalizes well is a difficult task that requires much computational power, large datasets, and a correct choice of a vast number of hyperparameters.

As there is no underlying theory for the influence of many hyperparameters, good settings can only be found by experimentation. Due to the large hyperparameter design space and computationally expensive training, only a tiny fraction of the search space can be explored manually. In consequence, hyperparameters are often set from experience and some limited experimentation. In this thesis, we will highlight the influence of the network architecture as an especially important hyperparameter and develop automated methods to optimize it. Neural network weights and architecture are strongly tied to each other, and there are different approaches to optimizing both.

We will first explore ideas stemming from neuroevolution research and scale them up to large-scale neural networks. This becomes possible through a GPU-accelerated implementation of an evolutionary algorithm for neural network training. We explore its performance on supervised learning problems and compare it to stochastic gradient descent. A similar evolutionary algorithm for neural network training is also demonstrated in a reinforcement learning setting. Both approaches show the ability of evolutionary algorithms to train large-scale neural networks, but disregarding the gradient in a supervised learning setting results in an inefficient algorithm.

Since we focus on supervised learning problems, we turn our attention to the field of neural architecture search, which employs stochastic gradient descent for the weight training. Our first evolutionary black-box neural architecture search algorithm incorporates a weight inheritance strategy that significantly reduces the amount of computation time that is necessary for the search process. This is an important achievement, as neural architecture search can take a prohibitive amount of time. In our second neural architecture search algorithm, we remove the black-box assumption and reduce computation time even further by optimizing weights and architecture simultaneously in a single pass. In an effort to reduce the required amount of training examples, we use this algorithm to perform multi-task learning and show strong performance compared to other multi-task learning algorithms.

Mathematical Notation

We will commonly need to refer to the set of integers up to a specific number and define the following notation for natural numbers $n \in \mathbb{N}$: $[\![n]\!] = \{1, \dots, n\}$.

Deep learning makes extensive use of tensors, which can be understood as multi-dimensional arrays for our purposes. They are necessary, because vectors and matrices cannot properly express the structure of our data in many cases. Whether one-, two-, or high-dimensional, we will denote these objects with a straight bold typeface like $\mathbf{a}$ or $\mathbf{A}$. In some cases, we will also use Greek symbols for vectors, matrices, or tensors. Similar to how a matrix with d_1 rows and d_2 columns is commonly denoted as $\mathbf{A} \in \mathbb{R}^{d_1 \times d_2}$, we will denote an n-dimensional tensor as $\mathbf{B} \in \mathbb{R}^{d_1 \times \dots \times d_n}$, where the dimensions have $d_1, \dots, d_n$ components respectively.

Components of a tensor will be indexed by subscript just like for matrices, i.e. $\mathbf{A}_{i,j}$ is scalar if $\mathbf{A}$ is a matrix, and $\mathbf{B}_{i_1,i_2,i_3}$ is scalar if $\mathbf{B}$ is a three-dimensional tensor. If the number of indices is less than the number of dimensions, a slice is returned. For example, $\mathbf{a} = \mathbf{A}_i \in \mathbb{R}^{d_2}$ refers to the i-th row vector in $\mathbf{A} \in \mathbb{R}^{d_1 \times d_2}$. Indexing the resulting vector at j returns the scalar $\mathbf{a}_j = \mathbf{A}_{i,j}$. The process is analogous for tensors so that $\mathbf{b} = \mathbf{B}_i \in \mathbb{R}^{d_2 \times \dots \times d_n}$ refers to the $(n-1)$-dimensional tensor that has the first dimension of $\mathbf{B} \in \mathbb{R}^{d_1 \times \dots \times d_n}$ fixed to index i, i.e. $\mathbf{b}_{j_1,\dots,j_{n-1}} = \mathbf{B}_{i,j_1,\dots,j_{n-1}}$.

We will also sometimes define sequences of tensors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ using subscripts. Whether a subscript indexes the tensor or refers to different tensors of the same sequence is made clear from context.

Contents

Chapter 1

Introduction

Deep learning has significantly advanced the state-of-the-art in many difficult learning problems. For example, large improvements over traditional computer vision algorithms have been achieved on the ImageNet image classification benchmark [106] using deep neural networks (DNN). In the domain of game playing, the deep reinforcement learning system AlphaZero [115] can play the perfect-information games Go, Chess, and Shogi at super-human level. AlphaStar [136] is competitive with professional players in the real-time computer strategy game StarCraft 2, and OpenAI Five [10] is competitive with professional players in the computer game Dota 2, which requires a team of five agents to cooperate. In natural language processing, significant advances in subfields like language modeling, translation, and speech recognition have been driven by deep learning as well. Many of these advances are due to new DNN architectures like the Transformer [134], and unsupervised pre-training methods like BERT [24]. As a final example, generative image modeling has become good enough to produce photo-realistic high-resolution images of human faces through the use of DNNs, for example with StyleGAN [55, 56] or PG-SWGAN [143].

Thanks to mature deep learning frameworks such as PyTorch [89] or TensorFlow [1], it has become easier than ever to experiment with deep learning models. This allows deep learning to be utilized in other fields of study, e.g. medicine. There is an increasing amount of applied research on medical applications that use deep learning for tasks like detecting diseases in various modalities, such as x-ray images [48], CT scans [90], MR scans [91], or histological samples [35]. Other uses include segmentation of cells [133], organs [57], and more, or providing information about surgical procedures from video [3, 37] that can be used in real-time computer assistance systems for surgeons. However, despite the barrier of entry being quite low, it is still difficult to take advantage of deep learning in real-world settings for a variety of reasons.

First, training a DNN requires comparatively expensive computational resources. Graphics processing units (GPU) are a necessity to train modern networks in a reasonable timeframe, because DNN models make extensive use of the parallel processing capabilities provided by GPUs. Due to the ever increasing amount of weights in DNNs, GPUs with enough memory to hold these weights are especially important. For this reason, many large models can only be trained on very expensive GPUs or

even specialized neural network hardware such as TPUs [53]. Inference, i.e. running a model just to get its outputs, is computationally cheaper but still difficult on low-end machines. This is a concern especially for edge computing, such as on mobile devices, and can only be achieved with specialized architectures that take such constraints into account. Nevertheless, because of the continued progress on hardware development, these problems will become less significant over time.

Next, DNNs require large amounts of data for training in order to achieve strong generalization. This is a consequence of the large amount of weights in DNNs. In general, increasing the size of a DNN leads to better results but only if enough data is available as well [86]. The supervised learning approach, which requires manually labeling training examples, is still the most useful for many practical problems. Consequently, it becomes work-intensive and costly to collect a big enough dataset to apply deep learning. We will see in Chapter 3 how transfer learning can be leveraged, and in Chapter 8 how multi-task learning can be leveraged to reduce the amount of labeled data that is necessary to train a DNN.

How well a DNN model performs will strongly depend on the many hyperparameters of deep learning systems, such as network architecture, weight initialization, optimizer, learning rate or learning rate schedule, regularization method and its hyperparameters, data augmentation, and many more. A hyperparameter of special interest is the network architecture because it is arguably among the most important factors that influence the final performance of a model. As we will see in Section 2.4, new architectures have continued to push the state-of-the-art in image classification, and the same is true in other domains as well, e.g. natural language processing where the Transformer [134] architecture is currently extremely prevalent and has mostly replaced older architectures.

It is difficult to experiment with hyperparameters to find a good configuration because the design space is extremely large and the training of DNNs is so expensive and time-consuming. To illustrate, consider the network architecture, which is a directed acyclic graph (DAG) with a number of nodes that depends on the granularity of the search. If the search happens on the coarse level of network layers, a small network architecture might contain 10 layers, i.e. nodes. There are already more than 4×10^{18} different DAGs with 10 nodes [46], and this is not yet considering the fact that network architectures contain different kinds of nodes, many of which also have their own hyperparameters. Accordingly, the search is rarely performed in the space of all DAGs and instead focuses on subspaces that are known to perform well. However, even these reduced search spaces are combinatorial and extremely large.

Therefore, hyperparameters are usually set from experience, and experiments are performed for a few limited configurations. This often amounts to re-using the most popular or recent architecture from literature and running a few experiments with different learning rate schedules. The result is very likely to be a sub-optimal configuration for any given problem. It would be desirable to have automated techniques

that search for hyperparameters (among them the architecture), which achieve the best possible training results. Such a system would lower the required level of expertise to employ deep learning systems, and make deep learning accessible even to people unfamiliar with cutting edge machine learning research. However, due to the aforementioned large size of the search space and the computational requirements of DNN training, hyperparameter optimization for DNNs is a difficult problem. We will tackle it in Part III of this thesis.

In summary, we have identified three problematic areas that can be addressed to further lower the barrier of entry to the deployment of deep learning based systems. We will outline three goals and identify them as (G1), (G2), and (G3) in order to refer back to them in the further text of the thesis and relate our work to these goals. The main goal is to automate the hyperparameter selection process in order to avoid this time-consuming, manual work that needs to be performed by machine learning experts (G1). We are only concerned with the architecture as a hyperparameter, since it has strong influence on the result and is difficult to select manually due to the graph-based structure and consequently large search space. With the use of automated search techniques to select the architecture, the computational requirements rise even further. Therefore, another goal is to reduce the amount of computation required during such a search process (G2). Finally, even with automated architecture selection, large amounts of labeled data are still essential. We make it our third goal to reduce the amount of training data necessary (G3).

Outline. In Sections 1.1 and 1.2 we introduce the fields of neuroevolution and neural architecture search respectively. They represent important work towards our main goal (G1) and form the basis of our own approaches. Then, in Section 1.3, we present an overview of the whole thesis.

1.1 Neuroevolution

Automatically learning neural network architectures is a long-standing goal that sparked the field of neuroevolution. Earlier neuroevolution works optimize network architectures together with the network weights using some form of evolutionary algorithm. The approaches can be divided into those that apply an existing general evolutionary method to the problem of concurrent weight and architecture optimization for neural networks, and those that developed specialized algorithms based on evolutionary principles. Publications from the first category include using a genetic algorithm [141], evolutionary programming [109], or covariance matrix adaptation evolution strategy [45] for neuroevolution. The main issue here is developing a good representation for the architecture in the framework of the chosen optimization method. Examples from the second category of specialized approaches are SANE [84], ESP [33], CoSyNE [32], and NEAT [123] among others. NEAT is a well-known neuroevolution algorithm that allows to grow neural networks starting with a minimal

network and expanding it through mutation and principled crossover between network graphs.

What all these approaches have in common is that they have predominantly been applied in small-scale reinforcement learning domains with low-dimensional inputs and outputs. This is understandable, because these algorithms operate on single graph nodes or edges and therefore do not scale very well. For example, NEAT has been most successful on problems that can be solved with very small neural networks on the order of tens of nodes and with a similar amount of inputs [122, 124, 140, 120]. In contrast, consider the ImageNet benchmark where current DNN methods use millions of nodes and work on images of 331×331 pixels, i.e. around five orders of magnitude more nodes and four orders of magnitude more inputs. Consequently, directly applying NEAT to neural architecture search on problems that are commonly solved by deep learning today is bound to fail. Methods based on individual nodes and weights will not be able to explore any significant part of the extremely large search space and will generate small, suboptimal solutions.

HyperNEAT [121] is a neuroevolution method building on NEAT that is supposed to alleviate this problem. However, the method fails to deliver good results even when applying it to easy high-dimensional supervised learning problems. Verbancsics and Harguess [135] test if HyperNEAT can evolve a DNN for image classification on MNIST. In theory, HyperNEAT should be well suited to re-discover a convolutional architecture, as the genotype is structured so that symmetries and repeated patterns are easy to express. However, HyperNEAT merely achieves 23.9 % test accuracy on MNIST (compared to around 99.9 % for gradient-based DNN methods) and does not rediscover convolutional structures. Even when the convolutional structure is enforced externally, it still only achieves 27.7 % test accuracy. Clearly, a different approach is necessary to bring the ideas of neuroevolution into the field of deep learning with its large-scale problems and networks.

1.2 Neural Architecture Search

Research in the field of neural architecture search (NAS) is the modern equivalent to neuroevolution. The main difference to neuroevolution is that the weights corresponding to a DNN architecture are trained with stochastic gradient descent (SGD) to exploit its efficiency on supervised learning problems. We will see in more detail in Chapter 5 how well SGD performs at weight training on supervised problems compared to evolutionary algorithms. In NAS, the method to optimize the architecture is independent of the gradient-based weight optimization, and there are many different approaches using reinforcement learning [154, 155, 12], evolutionary algorithms [70, 92, 5, 80, 117], Bayesian methods [69, 54, 51], or even random search [145]. Random search works surprisingly well if the search space is biased to contain many good solutions, which is usually the case with current NAS approaches, as the search spaces are informed

by hand-crafted architectures from recent machine learning research. In fact, there is research pointing out that many NAS algorithms on average perform the same as random search on such search spaces [149]. NAS algorithms are able to automatically find very strong architectures as we will see in Section 2.4, but the listed approaches have considerable downsides as well.

The typical approach is to treat NAS as a black-box optimization problem, i.e. the architecture optimization procedure creates candidate architectures, they are trained by SGD, and return a validation loss or similar measure, which is used to guide the optimization. This requires the training of many DNNs and is computationally very expensive. For example, Zoph and Le [154] use 800 GPUs for 28 days (22,400 GPU-hours) to perform neural architecture search on CIFAR-10, while Zoph et al. [155] are able to reduce these requirements to 500 GPUs for 4 days (2,000 GPU-hours). Understandably, an important goal is reducing these still enormous computational requirements.

This is one of the reasons why many recent approaches have shifted their perspective on the problem and do not try to perform black-box optimization any longer. Weight inheritance is one possible fix that speeds up training, and we will examine this idea in Chapter 7. Another important line of research deals with one-shot NAS algorithms that learn weights and architecture concurrently in a single training process. Thereby, they avoid the repeated training of DNNs that black-box NAS algorithms have to perform. We will work with such an algorithm based on natural evolution strategies in Chapter 8.

A prominent example of one-shot NAS that has spawned a lot of follow-up work is DARTS [71]. DARTS makes architecture search differentiable with a deterministic attention mechanism. It searches in the space of complete DAGs with four nodes, which represent input values for the operations that are represented as edges. The goal is to select between a number of different operations from a given set for each edge (including no operation, i.e. the edge is not supposed to exist) in order to determine the network architecture. The choice between operations on an edge connecting node i to node j is made differentiable by expressing it as a weighted sum over the outputs of all possible operations applied to the same input from node i:

$$v\left(j\right) = \sum_{f \in \Omega} w_{i,j,f} f\left(v\left(i\right)\right), \tag{1.1}$$

where $v\left(i\right)$ is the value at node i, Ω is the set of operations, and $w_{i,j,f}$ is the normalized coefficient for operation f on the edge between nodes i and j. By adjusting the coefficients, individual operations can be made more or less important for the result, effectively changing the architecture represented by the DAG.

The network architecture, i.e. coefficients for the weighted sum on every edge, and the network weights are both trained by SGD but in two different passes. First, weights are trained using a loss on the training set as usual, while the architecture coefficients

are kept fixed. Then, the architecture coefficients are trained by minimizing a loss on the validation set after a single virtual SGD step on the weights. This single SGD step stands as a proxy for a complete training with SGD, as the authors want to select the architecture that has the lowest validation loss after training with SGD. After training, each edge in the DAG can be assigned the operation with the highest coefficient. However, during training all operations are active at all times. This results in increased computational cost and GPU memory requirements but, at 36 GPU-hours for a search on CIFAR-10, is still vastly more efficient than black-box NAS algorithms.

AtomNAS [78] uses the same idea of a deterministic attention mechanism but modifies the search space. Instead of selecting which operation to apply to a whole input tensor, this method searches operations for each channel of the input, making it a lot more fine-grained than before but still more coarse than the older neuroevolution approaches. By pruning operations whose learned coefficients are almost zero during the training instead of after the training, the whole process gets accelerated as well.

ProxylessNAS [13] uses the same search space as DARTS and is mainly concerned with reducing GPU memory consumption and computation. This is achieved by sampling operations from a probability distribution instead of performing a weighted sum. The coefficients are learned using a method called BinaryConnect [21], which is a training method originally invented to train neural networks with binary weights.

In a similar vein, SNAS [146] also samples its operations from the so-called concrete distribution which allows to analytically derive the gradient w.r.t. the distribution parameters. "This renders SNAS a differentiable version of evolutionary-strategy-based NAS" [146].

Indeed, there are also one-shot NAS algorithms building on evolutionary strategies. Shirakawa et al. [114] define a probability distribution over the network architecture and perform an alternating optimization of weights using SGD and distribution parameters using natural evolutionary strategies. In their publication, the architecture is modeled by a Bernoulli distribution, i.e. they can express binary choices such as whether or not to skip a layer in a predefined architecture, which of two activation functions to use, or turning connections between layers on or off. This approach was extended to a more complete architecture search by Akimoto et al. [2] with a categorical distribution to model the architecture choices and a DAG-based search space like previous NAS approaches.

1.3 Thesis Overview

This thesis is split into three parts. In the first part we introduce the foundations of deep learning and demonstrate the concepts using an application to cancer detection from microscopic images. The second part deals with evolutionary DNN weight optimization on supervised and reinforcement learning problems. This is motivated by the goal to connect neuroevolution methods to modern, large network architectures.

Finally, the third part presents two NAS approaches. First, we reduce required computation using an evolutionary NAS algorithm with Lamarckian properties and second, we reduce both required computation and data using an evolutionary one-shot NAS algorithm in combination with multi-task learning.

We generally test our methods on image classification problems because they are a great test bed for deep learning. Images are high-dimensional data that pose difficult learning problems. Furthermore, many practical deep learning applications learn from supervised or semi-supervised image data, e.g. all the previously mentioned medical applications [48, 90, 91, 35, 133, 57, 3, 37].

1.3.1 Foundations of Deep Learning

The first part of this thesis introduces deep learning. After a general discussion of function approximation with DNNs, we focus on building blocks and architectures, since they are essential to the understanding of NAS methods. We then showcase a successful application of DNNs to cancer detection that resulted in third place in an international competition.

1.3.2 Evolutionary Weight Optimization

Literature has shown that applying classical neuroevolution methods like NEAT or HyperNEAT to large DNNs does not produce results competitive even with simple hand-designed DNNs. However, there has not been much research into connecting other evolutionary algorithms (EA) with modern DNNs. It is an interesting question if neuroevolution, i.e. architecture search without using SGD for weight training, is still possible with large architectures. Like other architecture search methods, this would tackle the main goal (G1) that we identified earlier. However, unlike existing NAS approaches, having a unified algorithm that trains weights and architectures using the same means would be an elegant approach that lends itself to easier analysis than systems that have multiple interacting optimization procedures.

Such a unified algorithm would have unique advantages stemming from properties of EAs as well. First of all, they are extremely easy to parallelize, which allows to reduce the wall-clock optimization time by exploiting parallel computational resources. Then, the black-box nature of EAs allows DNNs and their objective functions to be non-differentiable. This allows, e.g. for direct optimization of objectives like the F1-score. In contrast, SGD depends on the whole system being differentiable and can only optimize proxy loss functions such as the weighted cross-entropy instead of the F1-score itself. Finally, the different optimization behavior of EAs might find different kinds of optima than SGD. This can result in different properties, e.g. more robust reinforcement learning agents as hinted in [66].

As a preliminary step to neuroevolution, we aim to answer the simpler question: *Can we train a large DNN with fixed architecture with an EA and achieve competitive*

results to SGD? If this is not possible in a satisfactory manner, training DNNs with an EA while also simultaneously optimizing their architecture is unlikely to work. Therefore, to start out we look at weight training in isolation from an evolutionary optimization perspective.

In Chapter 5, we use a population-based EA to train a DNN on a supervised image classification problem. We build on previous work that created the limited evaluation EA [85] and massively scale up their approach, identify problems, and propose fixes. Our test problem is the MNIST image classification dataset, which is significantly more challenging than the datasets other publications have used to evaluate evolutionary approaches, despite being regarded as a very easy problem in the machine learning community. Our EA optimizes a DNN with around 100,000 weights, which is also significantly more than usually tackled with EAs. We find that it is indeed possible to achieve good training results with our EA, but disregarding the gradient in a supervised learning setting results in an inefficient algorithm compared to SGD.

Reinforcement learning is a problem setting where it is more difficult to exploit gradient information, and we demonstrate another EA approach on a benchmark of Atari games. In Chapter 6, we build on work by Such et al. [125] who train a DNN agent to play Atari games. We identify fitness noise as a significant problem and increase the algorithms robustness with more suitable parameter choices and a re-evaluation procedure for elites. This leads to improved scores and outperforms gradient-based approaches in some games.

In summary, EAs are definitely applicable to large-scale problems like the training of DNNs and can bring unique advantages. However, in a supervised learning setting, it does not seem worth it to disregard the gradient information as training with SGD is vastly more efficient. Since we are concerned with exactly these kinds of problems, we will turn our attention to NAS approaches that train weights by SGD while optimizing the architecture with EAs.

1.3.3 Evolutionary Neural Architecture Search

Despite the high efficiency of SGD compared to EAs for weight training, NAS algorithms that train weights with SGD are still extremely expensive. They either need to train thousands of networks or perform computationally expensive tricks to cast NAS as a differentiable problem. A natural question to ask is: *Can we make NAS more computationally efficient?* A more efficient NAS algorithm tackles the goals of automatic architecture search (G1) and reduced computational requirements (G2) that we identified earlier.

In Chapter 7, we use a standard NAS approach that performs black-box architecture optimization with an EA to find convolutional DNN architectures on image classification problems (G1). As a black-box approach, every fitness evaluation requires

the training of a DNN which makes the approach expensive to run. This is especially true for image classification problems, as the networks have increased computational requirements with increasing image size. To combat this, we use a weight inheritance scheme that can be regarded as an implementation of Lamarckian evolution, i.e. the inheritance of acquired traits. Instead of starting the training from random weights for every network architecture, the weights of a parent solution are used as a starting point to reduce training time. This results in a strong acceleration of the evolutionary architecture search process while keeping accuracy the same as without the weight inheritance process (G2). These claims are validated on four different image classification datasets.

In Chapter 8, we want to tackle all three goals of architecture search (G1), computational efficiency (G2), and data efficiency (G3) in a single algorithm. We present a one-shot NAS algorithm based on natural evolution strategies that concurrently optimizes weights and architecture. This means it is not necessary to train thousands of networks and therefore improves computational efficiency (G2). The approach is combined with deep multi-task learning, i.e. multiple supervised learning problems are learned at once in order to share knowledge and improve results on each task. When data for other similar tasks is already available, which is often the case, this tackles goal (G3), as good results can be achieved with less data for the target task. Our approach significantly improves upon previous deep multi-task learning algorithms on standard multi-task learning image classification benchmarks.

1.4 Contributions

This thesis connects evolutionary approaches to modern, large-scale deep learning in a resource-constrained setting. In this section, we list important contributions to scientific progress that stem from our EA applications to weight training, architecture search, and multi-task learning.

For weight training, we advance the scale of networks that can be optimized by an EA on a single machine with a single GPU to around 100,000 weights compared to 1,500 in our LEEA [85] baseline. Using our framework, we achieve the highest MNIST accuracy to date (to the best of our knowledge) for a neural network trained with an EA on a single machine and GPU. This demonstrates that EAs can be a viable alternative to SGD for neural network training even without enormous computational resources, and our open-source implementation opens the door for further research on this topic to find and exploit unique advantages that EAs offer for this problem.

For architecture search, we design an EA and search space to create an evolutionary NAS algorithm that finds architectures with high accuracy on image classification benchmark datasets while using only a single GPU and multiple orders of magnitude less computation than similar [80, 98] approaches. This is realized by integrating weight inheritance into the EA mutation operators and demonstrates that NAS with

large-scale networks is possible even in resource-constrained settings.

For multi-task learning, we provide a new perspective on configuring a deep MTL system by formulating the process as a multi-task architecture search in which weight sharing between task-specific networks is optimized. Using a hybrid NES and SGD optimization, our LWS algorithm can perform deep MTL without manually choosing an MTL architecture and achieves significantly improved accuracy over recent MTL algorithms [81, 102, 79] on the CIFAR-100 and Omniglot benchmark datasets.

Part I

Foundations of Deep Learning

Deep neural networks are a class of statistical models that are used extensively in this thesis. Good introductory books such as [34] exist already; therefore, we will focus on the aspects of DNNs that are important to the further work in this thesis. Chapter 2 will explain how DNNs are used and trained but places special emphasis on their building blocks and architecture. This will be useful for later parts of the thesis that deal with neural architecture search. Furthermore, the purely theoretical description of DNNs is supported by Chapter 3 with an exemplary application to blood cancer detection.

Chapter 2

Deep Neural Networks

The research on artificial neural networks begins in 1943 with the seminal work of McCulloch and Pitts [77] who develop a first computational model of neurons that can be connected in graphs to create neural networks. There are two main lines of research; one that strives for biological plausibility and explaining the human brain, and one that strives to use them as a tool for artificial intelligence. Note that these goals are not mutually exclusive, and this divide is merely a question of research focus. Research into the latter direction eventually creates Rosenblatt's perceptron [103] in 1958 along with a first algorithm that can actually learn the weights of the perceptron from data. In 1986, Rumelhart et al. [105] present the multilayer perceptron model along with the generalized delta rule for learning from data.

The multilayer perceptron is already very similar in concept to today's deep neural networks, and the learning algorithms in use today still build on the ideas presented by Rumelhart et al. [105]. Both types of artificial neural networks are graphs with weighted edges where each node performs some kind of operation on its inputs. In the classical multilayer perceptron model and its derivatives, this is a weighted sum over its inputs followed by the application of a non-linear activation function. Since then, the field has evolved, and now DNNs contain many different kinds of nodes with specialized functions. Since we make extensive use of DNNs throughout this thesis, it is important to have a solid understanding of their basic concepts.

Outline. In Section 2.1, we introduce DNNs as a class of models with very strong function approximation capabilities and discuss issues regarding supervised learning with DNNs. Next, we touch on the optimization process that is used to train networks in Section 2.2, as it is important to understand it to appreciate the differences between the standard gradient descent based training and evolutionary approaches. In Section 2.3, we give an overview of modern building blocks that are used to create network architectures. We review network architectures that achieved state-of-the-art results on the ImageNet benchmark in Section 2.4. This will show how network architectures created from the previously introduced building blocks evolved over time. The more recent instances of these architectures are in common use today to solve a wide variety of problems.

2.1 Function Approximation

Deep neural networks are a powerful class of statistical models for function approximation. Such a model is defined by its network architecture and network weights. The architecture is a directed acyclic graph of operations applied to a network input, and each operation may involve (usually real-valued) weights that determine the operation's output. Conceptually, there is no difference between deep and shallow neural networks, but DNN is the term used for the modern network instances whose architectures are made up from many different computational layers so that there is a long computational path from input to output.

Neural networks can approximate any continuous function on a compact[1] domain to any desired accuracy, given that an appropriate architecture is chosen and the weights are set correctly. This fact was proven early on for two-layer dense networks by Hornik et al. [42]. More recently, similar proofs have been developed for Lebesgue integrable functions using modern architectures, such as residual networks [68] or ReLU-networks with different width and depth bounds [74]. This so-called universal function approximation property of neural networks can be used in various ways. In this thesis, we are mostly concerned with supervised learning, i.e. we use the network to approximate a given mapping between data and labels as best as possible.

Thanks to the universal function approximation property [42, 68, 74], any finite training dataset can be perfectly interpolated by DNNs, given that the network's model complexity is high enough and follows the restrictions outlined by the referenced proofs. For example, an interesting demonstration of this fact by Zhang et al. [151] shows that large DNNs with sufficiently high model complexity are able to perfectly fit random datasets.

Model complexity is difficult to formally define for DNNs, but it depends on the network architecture, which, among other things, determines the amount of weights. Recent research suggests that the training process and dataset labeling also play a role in characterizing model complexity of DNNs [86]. However, rather than finding weights that perfectly interpolate the training set, we are actually interested in weights that generalize well to an unknown test set that follows the same distribution as the training set. Only this property makes DNNs useful in practical predictive applications.

The generalization error of statistical models has been studied extensively and can be decomposed into model variance and bias [50]. The model variance quantifies how much its predictions change if the dataset changes, and it increases with model complexity. The model bias quantifies how much error is introduced by the model being too simple to represent the true data distribution and decreases with model complexity. Depending on the relative change between variance and bias as the model complexity increases, the total generalization error changes as well. It usually follows a U-shaped curve, decreasing first and then increasing again, i.e. additional model

[1] e.g. any finite topological space, or closed and bounded subsets of n-dim. Euclidean space.

complexity hurts generalization after some point, which is known as overfitting.

This is in contrast with empirical observations about DNNs, which can contain vast numbers of weights and still achieve very good generalization. This phenomenon was investigated by [118, 8, 86] and has been named deep double descent. The observation is that the classical U-shaped error curve is observed initially, but when the model complexity is increased even further, the generalization error begins to decrease again.

Since extremely large models are preferred according to these results, classical neuroevolution on the basis of single nodes is hardly applicable anymore. Modern DNNs often operate in this regime of high model complexity, which in practice means that the networks have a high number of weights and an appropriate architecture so that they achieve nearly zero training error. As a consequence of these massive model complexities, it seems more appropriate to describe DNN architectures in terms of building blocks that each already contain a large number of weights instead of individual neurons. This is the same view that modern NAS approaches have adopted as well.

2.2 Parameter Optimization

After defining a network architecture, the DNN model has to be trained on a dataset by optimizing its weight values. As mentioned earlier, we focus on supervised learning from a labeled dataset. More formally, we have a dataset

$$\left\{ \left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)} \right) \right\}_{i=1 \ldots N} \subseteq \mathcal{X} \times \mathcal{Y} \tag{2.1}$$

of tuples with datapoints $\mathbf{x}^{(i)}$ and their associated ground truth label $\mathbf{y}^{(i)}$. We define a DNN model $f\left(\theta, \mathbf{x}\right) : \Theta \times \mathcal{X} \rightarrow \mathcal{Y}$ that uses weights θ from the weight space Θ to map from the input space $\mathcal{X}$ to the label space $\mathcal{Y}$.

For a formal description we will assume that $\mathbf{x}^{(i)} \in \mathcal{X}$ and $\mathbf{y}^{(i)} \in \mathcal{Y}$, but this assumption is violated in practice. In actual DNN implementations, all calculations are performed on batches of data for efficiency reasons. Therefore, both input and label tensors would be expanded with a batch-dimension on their first axis before being used as input and target for the DNN. Furthermore, the labels $\mathbf{y}^{(i)}$ must not necessarily be in the same format as the network output. See for example Equation 2.7 for a loss function that expects labels to be integers while the network output is a vector.

The goal is to set the DNN weights θ so that a loss function $\mathcal{L}\left(\theta\right) : \Theta \rightarrow \mathbb{R}$ is minimized. The loss function expresses the approximation error between network outputs $f\left(\theta, \mathbf{x}^{(i)}\right)$ and ground truth labels $\mathbf{y}^{(i)}$ as a function of the network weights θ. The goal of training the DNN can now be expressed as the optimization problem

$$\theta^* = \arg\min_{\theta} \mathcal{L}\left(\theta\right), \tag{2.2}$$

i.e. finding weights θ^* that minimize the approximation error on the dataset. Note that we assume the network architecture to be fixed beforehand. Therefore, the error $\mathcal{L}(\theta)$ is solely determined by the network weights. However, as we will see in Section 2.4, the architecture has significant influence on the network error and should be chosen to fit the dataset. Nevertheless, the optimization of architecture as well as weights is much more difficult and completely avoided in the standard training setting that we explain now. We will later see in Part III how we can extend the optimization problem to included the network architecture as well.

Loss functions aggregate the loss over a batch of B examples. The batch size B can in theory be chosen equal to N, i.e. the aggregation would be performed over the whole dataset, but this is prohibitively expensive in terms of computation. Let us define

$$\mathcal{L}(\theta) = \frac{1}{B} \sum_{i=1}^{B} \ell\left(f\left(\theta, \mathbf{x}^{(i)}\right), \mathbf{y}^{(i)}\right), \tag{2.3}$$

where $\ell(\widehat{\mathbf{y}}, \mathbf{y}) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ returns the error between the network output $\widehat{\mathbf{y}}$ and the label $\mathbf{y}$. Its definition depends on the structure of the label space $\mathcal{Y}$ and the problem that needs to be solved. For example, the mean-squared error could be expressed as

$$\ell_2(\widehat{\mathbf{y}}, \mathbf{y}) = \|\widehat{\mathbf{y}} - \mathbf{y}\|_2^2 \tag{2.4}$$

and is used for various regression problems with real-valued label spaces of arbitrary dimensionality. Since we concern ourselves mostly with image classification in this thesis, the negative log-likelihood or cross-entropy loss

$$\ell_{\text{pr.xent}}(\widehat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{C} \mathbf{y}_i \log(\widehat{\mathbf{y}}_i) \tag{2.5}$$

will be used very often. In this case, C is the total number of classes and the label space

$$\mathcal{Y} = \left\{ \begin{pmatrix} p_1 & \dots & p_C \end{pmatrix} \,\middle|\, p_i \in [0,1] \wedge \sum_{i=1}^{C} p_i = 1 \right\} \tag{2.6}$$

contains vectors of class probabilities. Since all but one component in the ground truth label vectors are zero (one-hot encoding), the sum always has just a single non-zero summand. However, in implementation, DNNs for classification are built to output real-valued class scores instead of probabilities and trained with the following equivalent formulation of the cross-entropy loss

$$\ell_{\text{xent}}(\widehat{\mathbf{y}}, c) = -\log\left(\frac{\exp(\widehat{\mathbf{y}}_c)}{\sum_{j=1}^{C} \exp(\widehat{\mathbf{y}}_j)}\right) = -\widehat{\mathbf{y}}_c + \log \sum_{j=1}^{C} \exp(\widehat{\mathbf{y}}_j), \tag{2.7}$$

where c is the index of the target class. Equation 2.7 is the negative log-likelihood loss combined with a softmax function to be able to treat the model output as class

probabilities. In this formulation, network output and label space are encoded differently, i.e. $\ell_{\text{xent}} : \mathbb{R}^C \times [\![C]\!] \to \mathbb{R}$, but it is more numerically stable and therefore preferred. Furthermore, it is always possible to convert between the integer class index and one-hot encoding of labels. It is easy to see that ℓ_{xent} is minimized when the network output for the target class c is large compared to all remaining outputs.

In order to minimize $\mathcal{L}(\theta)$, it is differentiated w.r.t. θ and then stochastic gradient descent is performed. Given that modern datasets are very big, it would be very costly to use the whole dataset to calculate both $\mathcal{L}(\theta)$ and $\nabla_\theta \mathcal{L}(\theta)$. Instead, a random batch of data is sampled and used for both calculations. The difficulty then lies in the efficient computation of $\nabla_\theta \mathcal{L}(\theta)$, even for a small batch of data, due to the large number of gradients that have to be calculated. This is achieved by the backpropagation algorithm [105], which is a special case of reverse-mode automatic differentiation [7]. Using the chain-rule and by caching intermediate results, the derivative w.r.t. every component of θ can be efficiently determined. With it, the weights can be updated in the direction of lower loss by setting

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta), \tag{2.8}$$

where η is a small real number called the learning rate. This process of repeatedly sampling a batch of data, propagating the data through the network to get the loss, performing reverse-mode differentiation to get the derivative, and then updating the weights is the essence of training a DNN.

However, many improvements to the simple stochastic gradient descent step can be made. This has led to the development of numerous so-called optimizers. An extremely popular optimizer with good empirical performance is Adam [59]. Similar to the way batch normalization layers (see Subsection 2.3.5) keep running statistics about the first and second moment of the data flowing through the network, Adam keeps running statistics about the gradients' first and second moments. Using this information, the learning rate is scaled individually per weight, increasing with either large first moments or small second moments. This leads to accelerated convergence compared to standard SGD.

2.3 Building Blocks

Classical multilayer perceptrons consist of nodes that have weighted connections to and from other nodes and compute functions of their scalar inputs. This effectively is a computational graph where each graph node corresponds to a node in the multilayer perceptron. We could describe DNNs in the same way, but this approach is not well suited to effectively describe the massive DNN models that are in use today. Instead of operating on scalar values, the computational graph is expressed in terms of building blocks that operate on tensors. In addition to being easier to handle, such a

representation matches the computational model of GPUs that are used to accelerate DNN computations.

Figure 2.1 shows a simple DNN architecture (computational graph) for image classification that consists of convolutional layers, batch normalization layers, ReLU activation functions, average-pooling layers, and dense layers. In our example, the input to the network is a 32×32 pixel RGB image. In order to use an image as the input for a DNN, it is represented as a real-valued three-dimensional tensor, here $\mathbf{x} \in \mathbb{R}^{3\times32\times32}$, with the first axis indexing the color channels and the remaining two axes indexing the spatial position. If the image was grayscale instead of RGB, we would have an input tensor $\mathbf{x} \in \mathbb{R}^{1\times32\times32}$.

The input data flows through the computational graph and is transformed by each of the layers before being output after the final layer. Notice how its spatial structure is preserved throughout the first part of the network, because all of the layers there are equipped to deal with spatial data. However, the dense layers accept vector-valued inputs only, so that the tensor has to be reshaped, and the spatial structure is lost. Looking at the graph as a whole, it defines a function that transforms an image to a real-valued vector that can be interpreted as scores for a classification problem. We will now go over each of the contained building blocks in detail in the next sections.

2.3.1 Dense Layer

The dense layer, also called fully connected or linear layer, is a conceptually very simple block that takes an input vector $\mathbf{u} \in \mathbb{R}^{d_1}$ and applies a linear function to create its output

$$\mathbf{v} = \mathbf{W}\mathbf{u} + \mathbf{b} \tag{2.9}$$

with weights $\mathbf{W} \in \mathbb{R}^{d_2\times d_1}$ and bias $\mathbf{b} \in \mathbb{R}^{d_2}$. Since it is a linear function of its inputs, a neural network made only from dense layers can only fit linear functions. To approximate arbitrary functions, it is necessary to intersperse the network with non-linear activation functions.

2.3.2 Activation Functions

Activation functions are applied element-wise to tensors of arbitrary dimensionality in between linear layers to provide the non-linearity that is necessary for universal function approximation. Traditionally, the sigmoid and hyperbolic tangent functions have been used, but both have problems related to vanishing gradients during optimization, because their gradients approach zero for large positive or negative values. Instead, rectified linear units

$$\mathrm{ReLU}\,(x) = \max\,(0, x)\,, \tag{2.10}$$

which have been shown in [31] to improve DNN training, are now the activation function commonly found in literature.

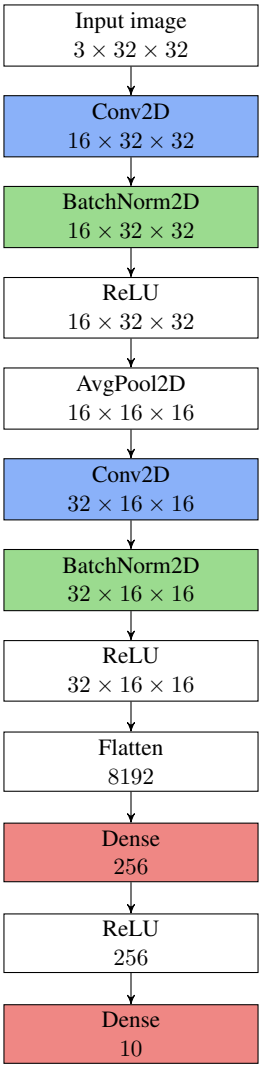

Figure 2.1: A simple convolutional neural network architecture for classification of 32×32 pixel three-channel images into 10 classes. For the first four nodes in the graph, the data has spatial structure with a height and width of 32 pixels. For the next four nodes in the graph, the data still has spatial structure but has been downsampled to a height and width of 16 pixels. Finally, in the last part of the graph, the data has lost its spatial dimensions and is now a vector. The colored building blocks contain trainable weights, while the rest of the blocks perform fixed functions.

2.3.3 Convolutional Layer

To motivate the use of convolutional layers, consider the case of applying a dense layer to an image input. Even moderately sized images of e.g. 100×100 pixels with a single color channel result in an input vector with 10,000 elements when flattened. This leads to an extremely high number of weights in the linear layer and consequently high resource requirements and higher model complexity, for which more data is necessary to achieve a good generalization error.

However, this situation can be improved by considering desired invariance properties. Imagine the input image to be embedded in a larger black image. Ideally, a classification network would output the same scores, no matter at which position the image is embedded in the larger image. Exactly this translation invariance is exploited by convolutional layers.

The convolutional layer can be understood as a special case of the dense layer that is applied to local chunks of structured inputs while sharing weights between applications at different chunks. Even though it can be expressed in arbitrary dimensionality, we will now describe the spatial (two-dimensional) convolution, which accepts a three-dimensional input tensor $\mathbf{u} \in \mathbb{R}^{c_1 \times h_1 \times w_1}$ with one channel dimension and two spatial dimensions. Such an input could represent a c_1-channel image of $h_1 \times w_1$ pixels, but generally c_1 will be much larger than the typical three channels for a color image. These tensors are also called a feature map, because the values $\mathbf{u}_a$ in each channel a can be understood as indicators for the presence or absence of a channel-dependent feature at each spatial position.

The output will be another three-dimensional tensor $\mathbf{v} \in \mathbb{R}^{c_2 \times h_2 \times w_2}$ and is created by first performing a valid cross-correlation of $\mathbf{u}$ with a kernel $\mathbf{k} \in \mathbb{R}^{c_2 \times c_1 \times h_k \times w_k}$, and then adding a bias $\mathbf{b} \in \mathbb{R}^{c_2}$ to all elements in each output channel. The number of output channels c_2 and kernel size $h_k \times w_k$ are hyperparameters of the convolutional layer. The cross-correlation, also called sliding dot-product, is used in practice instead of a true convolution. Since cross-correlation and convolution are interchangeable if the kernel is appropriately transformed, there is no functional difference for DNNs as kernels are learned from data anyways.

The kernel $\mathbf{k}$ expresses a multi-channel cross-correlation with c_2 output channels that are calculated independently. For each output channel with index $b \in [\![c_2]\!]$, $\mathbf{u}$ is convolved with $\mathbf{k}_b$ by sliding a three-dimensional window of size $c_1 \times h_k \times w_k$ over the spatial dimensions of $\mathbf{u}$. The stepsizes h_s and w_s with that the volume is moved along the spatial axes are called stride and are hyperparameters. In the simplest case, both are set to one. This sliding window process results in a number of chunks arranged in a grid that are of the same size as the kernel $\mathbf{k}_b$. For every chunk, an elementwise multiplication between it and the kernel $\mathbf{k}_b$ is performed before summing all elements. By performing this dot product for every chunk and collecting the resulting scalar values, we create a grid of values that make up $\mathbf{v}_b$. Finally, the bias $\mathbf{b}_b$ is added to

every element of $\mathbf{v}_b$. More formally, the output elements are given by

$$\mathbf{v}_{b,i,j} = \sum_{a=1}^{c_1} \sum_{m=1}^{h_k} \sum_{n=1}^{w_k} \mathbf{u}_{a,ih_s+m-1,jw_s+n-1} \mathbf{k}_{b,a,m,n} + \mathbf{b}_b, \tag{2.11}$$

where i, j must be chosen so that all indices remain valid. As a consequence, the spatial extent of the output will be smaller than the spatial extent of the input, i.e. $h_2 < h_1$ and $w_2 < w_1$. To preserve the spatial dimensions of the input, we can use a padded input tensor with spatial dimensions of size $h_1 + h_k - 1$ and $w_1 + w_k - 1$. Zero padding is very common and can be easily formalized by assuming $\mathbf{u}$ to be zero-valued at all invalid indices.

Preserving the spatial extent is not always necessary or even desired. In fact, convolutions can also be used to explicitly downsample the spatial dimension by choosing a stride larger than one. Using $h_s = w_s = 2$ and zero padding, as described before, results in an output of spatial extent $h_2 = \frac{h_1}{2}$ and $w_2 = \frac{w_1}{2}$, independent of the chosen kernel size $h_k \times w_k$.

As mentioned before, the convolutional layer and dense layer are very similar. In fact, a convolutional layer with a kernel as big as its input ($h_k = h_1, w_k = w_1$) is equivalent to a dense layer, whereas a convolutional layer with a 1×1 kernel applies the same linear transformation at each spatial position in the input. This can easily be verified using Equation 2.11. Just like with dense layers, non-linear activation functions are necessary to fit non-linear functions with networks made from convolutional layers.

2.3.4 Pooling

Pooling layers are employed to aggregate data and, just as for convolutional layers, we describe spatial (two-dimensional) pooling, which accepts a three-dimensional input tensor $\mathbf{u} \in \mathbb{R}^{c \times h \times w}$. There are different types of pooling layers that use different aggregation functions. Most commonly, average- and maximum-pooling are found, which apply the respective function to chunks of the input data. The process is very similar to the convolutional layer in that a window with a specified kernel size $h_k \times w_k$ and strides h_s, w_s is slid over the input tensor $\mathbf{u}$. However, the window is now two-dimensional and applied to each channel separately. Naturally, instead of performing a dot product, the aggregation function is applied. More formally, the output elements are given by

$$\mathbf{v}_{a,i,j} = \text{aggregation}\Big\{ \mathbf{u}_{a,ih_s+m-1,jw_s+n-1} \Big| m \in [\![h_k]\!], n \in [\![w_k]\!] \Big\}, \tag{2.12}$$

where i, j must be chosen so that all indices remain valid.

The reason to apply pooling is two-fold. First of all, the effective receptive field of following convolutional layers is increased because a single element of $\mathbf{v}$ now contains

aggregated information about a $h_k \times w_k$ region of its input $\mathbf{u}$, even if this aggregation is lossy. Secondly, the smaller spatial extent reduces the amount of computation and memory required for following convolutional layers.

2.3.5 Batch Normalization

Batch normalization [47] is a layer that reduces the difficulty of training DNNs that have many sequential layers by controlling the first and second moment of its output distribution. It can be defined for arbitrary dimensions, but again we focus on the case with a three-dimensional input and output tensors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{c \times h \times w}$. In contrast to the other layers that have been introduced so far, batch normalization layers not only have learned weights but also keep running statistics that are directly estimated from their input data. In particular, the mean μ_a and variance σ_a^2 of each input channel $\mathbf{u}_a$ are tracked independently for all $a \in [\![c]\!]$. Every time that data is passed through the layer, the mean and variance for each channel in the current batch are calculated:

$$\mu_a = \frac{1}{hw} \sum_{i=1}^{h} \sum_{j=1}^{w} \mathbf{u}_{a,i,j} \tag{2.13}$$

$$\sigma_a^2 = \frac{1}{hw} \sum_{i=1}^{h} \sum_{j=1}^{w} \left(\mathbf{u}_{a,i,j} - \mu_a\right)^2 . \tag{2.14}$$

If the batch normalization layer works in batch mode, these estimates are used directly in the following calculations. Otherwise, an exponential moving average is calculated from the previous estimate and the batch estimate. The resulting values are used to normalize the layer input $\mathbf{u}$ to have a mean of zero and variance of one:

$$\widehat{\mathbf{v}}_a = \frac{\mathbf{u}_a - \mu_a}{\sqrt{\sigma_a^2 + \epsilon}}, \tag{2.15}$$

where ϵ is a small constant for numerical stability. In order to allow the batch normalization layer to represent an identity transform, a linear function with weights $\gamma, \beta \in \mathbb{R}^c$ is applied to the normalized result to get the final output:

$$\mathbf{v}_a = \gamma_a \widehat{\mathbf{v}}_a + \beta_a. \tag{2.16}$$

There are strong empirical results showing the effectiveness of batch normalization to improve DNN training and it has become a standard component in DNNs. Generalization is improved compared to DNNs without batch normalization, and a wider range of learning rates can be employed.

Originally, Ioffe and Szegedy [47] related its success to the phenomenon of internal covariate shift. During training of a DNN, the input distribution to each layer constantly shifts as a result of the weights of all previous layers changing. This complicates the training process and is supposedly avoided by batch normalization.

Table 2.1: Error rates on the ImageNet dataset for different DNN architectures. [1]Results are achieved using ensembles of DNNs. [2]Results are achieved using a single DNN model.

Year	Method	Top-5 test err.[1]	Top-5 val. err.[2]	Top-1 val. err.[2]
2012	AlexNet [61]	16.4		
2014	ZFNet [150]	14.8		
2015	VGGNet [116]	6.8		
2015	GoogLeNet [128]	6.7		
2016	ResNet [39]	3.6	4.5	
2017	ResNeXt [144]	3.0	4.4	19.1
2018	NASNet [155]		3.8	17.3
2018	PNASNet [69]		3.8	17.1
2019	EfficientNet [129]		2.9	15.6

This explanation has been recently called into question by Santurkar et al. [108], because it is possible to inject noise in such a way into batch-normalized DNNs that they exhibit strong internal covariate shift and yet still perform just as well. The authors instead observe that batch normalization affects the smoothness of the loss landscape, effectively making gradients more predictive and explaining the empirically observed performance gains.

2.4 Modern Architectures

Having discussed the building blocks that are used to create neural networks, we now shift the focus on their assembly in a computational graph. Neural network architectures are a very important driver of progress in the field of deep learning, and this section will present substantial milestones. To measure how well an architecture performs, a difficult benchmark is necessary. This role is filled by the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that has been held annually from 2010 to 2017 [106]. It poses a difficult image classification task on a dataset of about 1.2 million large RGB images from 1000 different classes. Demonstrating state-of-the-art performance on this dataset is a good indicator for the general applicability of a network architecture to other visual tasks. Table 2.1 lists the network architectures that we will examine in this section and their best result on the ImageNet dataset. Note that over time, the metric of choice changed from top-5 to top-1 error. When using the top-5 error, an image is counted as classified correctly if its target class is among the top-5 predictions of the model. This metric got too easy as networks improved, and now top-1 error is usually reported.

One of the most influential publications that led to a resurgence of interest in DNNs is arguably the AlexNet paper [61] from 2012. From today's point of view, their network architecture is simplistic, but the paper popularized many ideas, such

as the ReLU activation function, which is still widely used today. The AlexNet architecture consists of a linear stack of 5 convolutional layers and 3 dense layers. Convolutional and dense layers (except for the last one) are followed by ReLU activations, and the convolutional layers are interspersed with max-pooling and local response normalization. The normalization layer is of their own design, and they do not use the previously described batch normalization layer, because it had not been invented yet. Using this architecture and an ensemble of five models, they achieve 16.4 % top-5 test error on the ILSVRC-2012 dataset. This DNN approach represents a significant improvement over the previous state-of-the-art approach, which achieved 26.2 % top-5 test error using classical computer vision techniques.

In 2013, ZFNet [150] is published. The authors visualize the convolutional kernels of AlexNet and slightly change the network design according to their observations. By changing the kernel sizes and strides of some convolutional layers, the top-5 test error of their DNN ensemble reaches 14.8 %. Even though the architectural differences to AlexNet are rather minor, a significant decrease in error is observed.

In 2015, VGGNet [116] is published. It still uses the same building blocks as AlexNet but significantly changes network depth and hyperparameters of each block. There are now 16 trainable layers in total, 13 of them convolutional and 3 dense, and the network is up to 512 channels wide. This yields a strongly improved top-5 test error of 6.8 % using an ensemble. An absolute improvement of almost 10 % compared to AlexNet was possible without any fundamental changes to the network but just by arranging existing components differently, increasing network depth, and setting layer hyperparameters like channel count appropriately. This illustrates how important the network architecture is.

Also in 2015, GoogLeNet [128] is published. Its main innovation are the so-called inception modules. The network consists of 3 convolutional layers, 9 inception modules, and 1 dense layer but has a total depth of 22 layers because of the inception modules. Each inception module contains four branches of differently parameterized convolutional layers that are shown in Figure 2.2. This presents a stark contrast to the previous architectures that only had a single path from input to output, whereas GoogLeNet has multiple. This new concept achieves 6.7 % top-5 test error using an ensemble and, while this isn't much of an improvement over the single path VGGNet, the concept of branching computation paths is found in all newer architectural designs.

The next major innovation happens in 2016 with the release of ResNet [39]. With the introduction of residual blocks, shown in Figure 2.3, it becomes possible to train extremely deep networks with hundreds of layers. A residual block (without the final activation function) computes the mapping $g(\mathbf{u}) = f(\mathbf{u}) + \mathbf{u}$, where $f(\mathbf{u})$ is a trainable mapping represented by three convolutional layers. In other words, the learning problem is modified so that the convolutional layers only have to approximate the difference between their input and a desired output. This also results in a multi-branch network architecture, although the second branch is just an identity function in

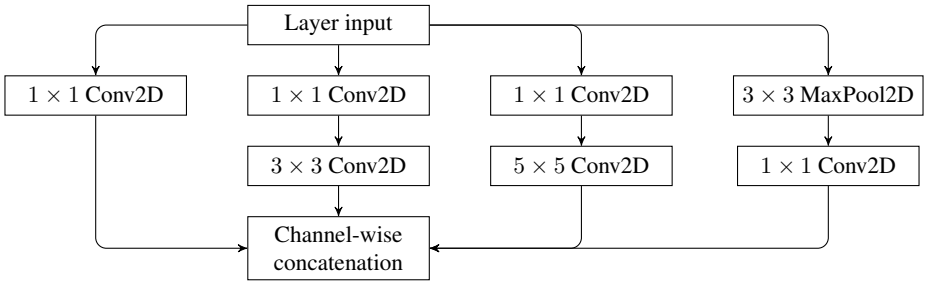

Figure 2.2: The branching micro-architecture of an inception module. Convolutions of different kernel sizes and a max-pooling layer are employed in parallel, and 1×1 convolutions are used for dimensionality reduction in the channel dimension.

most cases. Only when the number of input and output channels of the residual block differ is the identity function replaced with a 1×1 convolution to match the number of channels between $\mathbf{u}$ and $f(\mathbf{u})$. A residual network with 152 trainable layers is able to significantly beat the GoogLeNet architecture, achieving 3.6 % top-5 test error using an ensemble. Since ensembles of huge DNN models are not very practical, and the DNNs have improved so much over time, it has become common practice to report single model results on the ImageNet validation set. In the single model setting, ResNet-152 achieves 4.5 % top-5 validation error.

In 2017, ResNeXt [144] improves upon ResNet by re-using their general architecture but introducing grouped convolutions. Grouped convolutions split the input tensor into groups along the channel dimension and perform a convolution separately for each group. This is equivalent to many parallel convolutions in a branched architecture, as shown in Figure 2.4. Instead of only two branches per residual block, there are now many more, e.g. 32 convolutional branches and one identity branch in the ResNeXt instances proposed by the original publication. These networks achieve 3.0 % top-5 test error in an ensemble and 4.4 % top-5 validation error as a single model.

In 2018, the first architectures designed by algorithms beat human-designed architectures. NASNet [155] is created by performing neural architecture search on the much smaller CIFAR-10 dataset and transferring the design to ImageNet by scaling it up. The method searches for the design of two building blocks, called normal cell and reduction cell. The reduction cell always downsamples the input's spatial dimensions by applying operations with a stride of two, while the normal cell leaves the spatial dimensions unchanged. Both cells have a predefined graph structure, but the operations inside this structure are chosen by a reinforcement learning algorithm. The cells are then stacked to create architectures for different problems. This allows to search for well-performing cells on easier problems and then, by including more reduction cells, extending the architecture so that larger image inputs of other problems can be handled. When applied to ImageNet, NASNet achieves 3.8 % top-5 validation error with a single model. This is a significant improvement over ResNeXt, and at the same

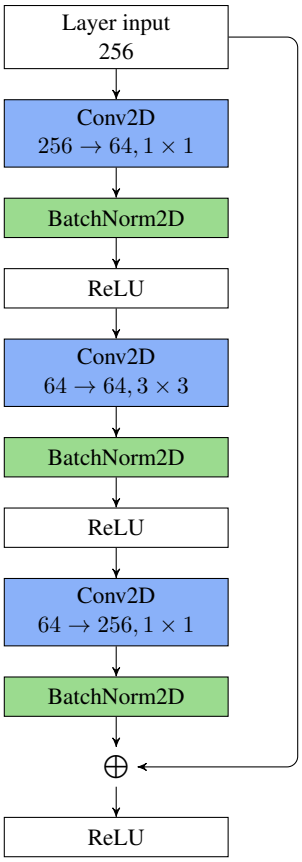

Figure 2.3: The branching micro-architecture of a residual block, here shown with 256 input channels and a 64 channel bottleneck. There is a skip-connection from the input of the residual layer to right before the final ReLU function. The input data is added element-wise to the output of the main convolutional branch. This particular residual block is the "bottleneck" variant that first downsamples the channel dimension with a 1×1 convolution before upsampling it again at the end with another 1×1 convolution. This approach is similarly found in inception modules.

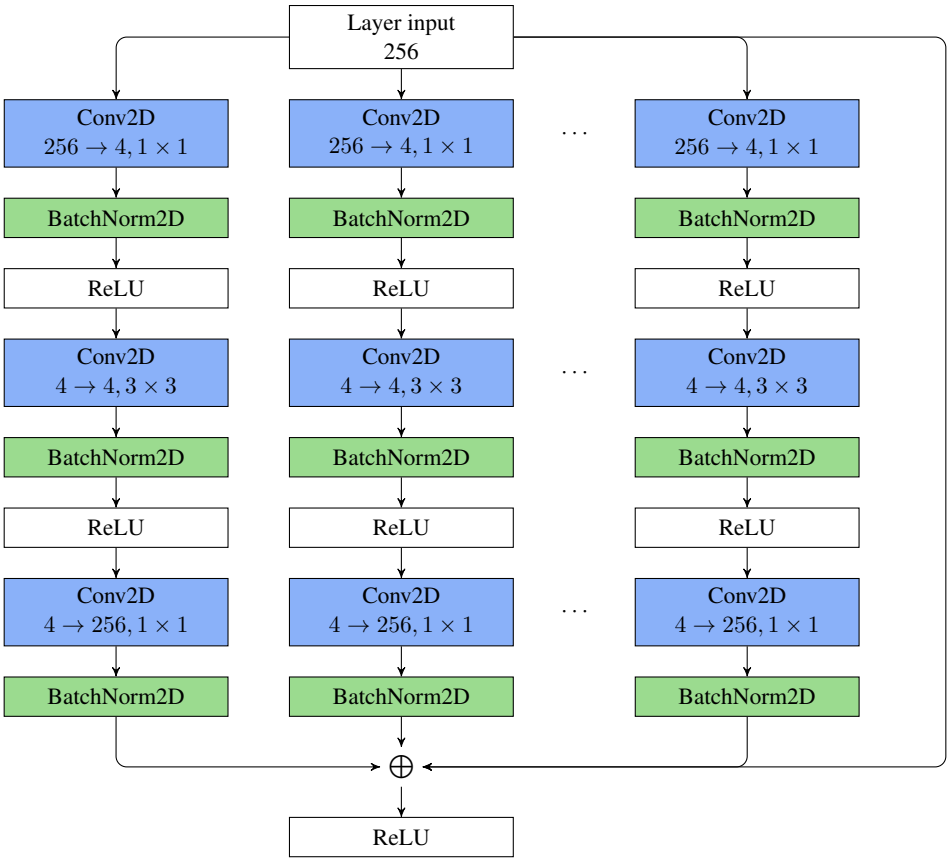

Figure 2.4: The branching micro-architecture of a residual block with grouped convolutions as used in ResNeXt. Just like for ResNet, the channel dimension is first downsampled and later upsampled again. However, differently from ResNet, there are a number of parallel paths with convolutions. This particular ResNeXt block shown here splits its 256 channel input into many groups of 4 channels using the 1×1 convolutions. For efficiency reasons actual implementations use grouped convolutions which perform an equivalent computation to what is shown here in a single operation.

time the network is computationally more efficient, because it performs about 25 % less multiply-accumulate operations. However, the downside is that the search process is extremely expensive, as it took 500 GPUs over 4 days to find these cells.

Still in 2018, PNASNet [69] is released. This model is also found by neural architecture search but uses sequential model-based optimization instead of reinforcement learning for the search. Their approach uses about five times less compute for the search process, and the resulting network performs just as well on ImageNet, also achieving 3.8 % top-5 validation error with a single model.

Finally, in 2019, an architecture named EfficientNet [129] is published. The authors investigate how network architectures should best be scaled up for more difficult problems. Many previous articles, both about hand-designed and automatically searched architectures, build their models on a small dataset and later scale it up for difficult problems like ImageNet by increasing the depth (number of layers), width (number of channels), or (spatial) resolution. However, usually this scaling happens in one of these dimensions only, whereas [129] argues that it is important to scale all of them in the right ratio to each other. This makes intuitive sense, as higher resolution images will need more depth and width to be processed. Using neural architecture search, they create a baseline architecture that they then scale up with their new approach. On ImageNet, the largest EfficientNet achieves 2.9 % top-5 validation error with a single model.

Seeing how neural architecture search helps push the state-of-the-art on the difficult ImageNet benchmark showcases its usefulness. The methods are generic so that they can also be applied in other domains, and this has already been done, e.g. for natural language processing [154, 117, 71]. The biggest downside of NAS algorithms so far is that they are still extremely resource intensive, even if this aspect has already been improved a lot since the field's inception. This can somewhat be alleviated by cleverly combining neural architecture search with other ideas and restricting the search space. We will look at two ideas for resource efficient neural architecture search in Part III.

For now, we hope that the discussion of different models in this section highlights how much of an impact the network architecture has on the model's error and that it is worthwhile to optimize it.

Chapter 3

Application to Blood Cancer Detection

In this chapter, we apply a DNN model to a real-world image classification problem to showcase the concepts that have been discussed on a theoretical level in the previous chapter. We will also work with fine-tuning to reduce the amount of data necessary for training. In the machine learning field, it is common to see public challenges that provide a real-world dataset and problem setting for competitors to solve. The IEEE International Symposium on Biomedical Imaging 2019 challenge on "Classification of Normal versus Malignant Cells in B-ALL White Blood Cancer Microscopic Images"[1] posed the problem of examining blood cell microscopic images for acute lymphoblastic leukemia (ALL).

Examining blood microscopic images for leukemia is necessary when expensive equipment for flow cytometry is unavailable. Automated systems can ease the burden on medical experts for performing this examination and may be especially helpful to quickly screen a large number of patients. We present a simple, yet effective classification approach using a ResNeXt convolutional neural network with squeeze-and-excitation modules. The approach was developed for the IEEE C-NMC challenge and achieved the third place in this international competition with a weighted F1-score of 88.91 % on the test set.

The results presented in this chapter are based on the following publication with accompanying source code:

- Jonas Prellberg and Oliver Kramer. Acute lymphoblastic leukemia classification from microscopic images using convolutional neural networks. In Anubha Gupta and Ritu Gupta, editors, *ISBI 2019 C-NMC Challenge: Classification in Cancer Cell Imaging*, pages 53–61, Singapore, 2019. Springer Singapore

- `https://github.com/jprellberg/isbi2019cancer`

Outline. In Section 3.1, we introduce the challenge, explain its significance, and present related work on ALL classification from microscopic images. Section 3.2 describes the dataset. The convolutional neural network, a ResNeXt-variant with

[1] https://biomedicalimaging.org/2019/challenges/

squeeze-and-excitation modules, that our approach is based on is presented in Section 3.3. In Section 3.4, we describe the data augmentation strategy, outline the training process, and show experimental results. Section 3.5 summarizes the chapter.

3.1 C-NMC Challenge

Acute lymphoblastic leukemia is a blood cancer that is characterized by the proliferation of abnormal lymphoblast cells, eventually leading to the accumulation of a lethal number of leukemia cells [96]. If ALL is diagnosed in an early stage, treatment is possible. Diagnosis is typically performed using a complete blood count and morphological analysis of cells under a microscope by a medical expert. Flow cytometry can replace this manual work but requires expensive equipment, which is not available everywhere. Therefore, automated systems that can perform diagnosis using comparatively low-cost microscopic images provide a great advantage. Bringing down the cost and labor required for such a test allows to have this test done for many more patients than before.

Previous work on automated ALL diagnosis from images can roughly be divided into more recent approaches that use convolutional neural networks (CNN), either as feature extractors or in a fine-tuning setting, and older approaches that use handcrafted features.

In the latter category, Putzu and Ruberto [97] work with the ALL-IDB [63] dataset and classify a number of hand-crafted features like area, compactness, roundness, and area ratio between cytoplasm and nucleus with a support vector machine. Mohapatra et al. [83] use an ensemble of naive Bayes, k-nearest neighbors, multilayer perceptron, and support vector machine for cell classification, while Madhloom et al. [76] rely on a k-nearest neighbors classifier alone. Both publications work with different private blood cell image datasets.

Publications working with CNNs include Rehman et al. [100], who classify ALL subtypes on a private dataset of 330 images using a pre-trained AlexNet and fine-tuning. Shafique and Tehsin [113] classify ALL subtypes on ALL-IDB augmented with 50 private images, also using a pre-trained AlexNet and fine-tuning. Vogado et al. [137] classify ALL on ALL-IDB using a number of different pre-trained CNNs as fixed feature extractors. From these CNN features, the most informative ones are selected using principal component analysis, and finally classification is performed with an ensemble of support vector machine, multilayer perceptron, and random forest.

In these cases, pre-training and fine-tuning refer to the process of first training the CNN on the ImageNet dataset and using the resulting trained weights as the starting point for training on the much smaller cell dataset. Due to the pre-training, many generally applicable kernels will develop in the first few convolutional layers, e.g. kernels that react to edges or certain colors [148]. These can be re-used on the target dataset, even if it is quite different from ImageNet. When the target dataset is small,

Table 3.1: Composition of the CNM-C dataset. The test set is unreleased and can only be evaluated against using an online service, so its composition is unknown.

Dataset part	ALL subjects	Normal subjects	ALL cells	Normal cells
Train	47	26	7272	3389
Prelim. test	13	15	1219	648
Final test	9	8	?	?

this process works best with carefully tuned learning rates or by excluding the first few convolutional layers from optimization (also called freezing) in order to mostly preserve the kernels from the pre-training step.

All mentioned publications about ALL classification report good results, but it is hardly possible to compare them, because the private datasets are unavailable and, even on the public ALL-IDB dataset, researchers employ their own evaluation procedures. Furthermore, the employed datasets are small in all cases, containing a couple hundred images at most. For example, ALL-IDB2 [63] contains only 260 images of white blood cells, which makes the dataset too small to properly take advantage of recent deep learning approaches, as can be seen by the ubiquitous use of pre-training.

To further research on ALL detection or classification, large public datasets are necessary to compare different approaches and track the state-of-the-art. In 2018, the C-NMC challenge dataset with more than 10,000 training images and a separate test set of normal B-lymphoid precursors and malignant B-lymphoblasts has been released as a challenge open to the public [35]. The large size of this new dataset allows to create improved classifiers based on deep neural networks and also provides a more reliable comparison of competing approaches.

3.2 Dataset Description

The C-NMC dataset contains images of white blood cells taken with a microscope from blood samples of 154 individual subjects, 84 of which exhibit ALL. Blood cells from different people will have slightly different characteristics so it is important to achieve generalization over subjects. Table 3.1 provides a detailed breakdown of the number of subjects and cells in three splits of the dataset. The dataset is imbalanced with about twice as many ALL cells as normal cells, which needs to be considered during the training of a DNN. It has been split into training set, preliminary test set, and final test set, because the competition featured an elimination stage using the preliminary test set. After this point, labels for the preliminary test set were released, and the examples became additional training data.

Figure 3.1 shows four example images from the dataset. Each image has a resolution of 450×450 pixels and contains only a single cell as a consequence of preprocessing

(a) (b) (c) (d)

Figure 3.1: Example images taken from the C-NMC training set. The black background has been made transparent for these figures. **(a)** ALL cell. **(b)** Normal cell. **(c)** ALL cell with part of the cell cut off due to an imperfect segmentation. **(d)** Normal cell with superfluous background due to an imperfect segmentation.

steps applied by the dataset authors: An automated segmentation algorithm has been used to separate the cells from the background. Each pixel that was determined not to be part of the cell is colored completely black. However, since the segmentation algorithm is not perfect, there are instances where parts of the cell are inadvertently colored black or superfluous background is included. Additionally, all images have been preprocessed with a stain-normalization procedure that performs white-balancing and fixes errors introduced due to variations in the staining chemical [36].

Distinguishing the cells is extremely difficult, as they appear similar morphologically. Instead of looking at individual cells, medical experts rely on domain knowledge for the identification of cancer in microscopic images, such as that cancer cells proliferate in an unrestricted fashion and therefore appear in large numbers. Machine learning techniques, however, are able to pick up on the slight differences between cells to identify them even in early stages when very few cells are abnormal.

3.3 Network Architecture

As we have seen in Section 2.4, image classification benchmarks have driven the creation of many powerful convolutional neural network architectures. We pick one of the recent top-performing networks, a ResNeXt-50 [144], as our base model.

The network consists of five convolutional stages with spatial downsampling by a factor of two in between stages, followed by global average-pooling and a dense layer. Each stage is made from stacked building blocks, each of which computes a function of the form

$$\mathbf{v} = \mathbf{u} + \sum_{i=1}^{K} f_i\left(\mathbf{u}\right), \tag{3.1}$$

where K is called the cardinality that controls the number of parallel paths in a block. The complete layout of such a block is depicted in Figure 2.4. Its parallel paths are com-

puted by functions $f_i(\mathbf{u})$, which project the input tensor $\mathbf{u}$ into a lower-dimensional space, transform it, and project back into a space of the original dimensionality. They are implemented using a sequence of 1×1, 3×3, and 1×1 convolutional layers which are interspersed with ReLU functions and batch normalization layers.

The ResNeXt architecture can be augmented with the squeeze-and-excitation operation introduced in [43], which has been shown to improve performance on image classification benchmarks. In order to do so, the block function is modified to

$$\mathbf{v} = \mathbf{u} + s\left(\sum_{i=1}^{K} f_i(\mathbf{u})\right), \tag{3.2}$$

where $s : \mathbb{R}^{c\times h\times w} \to \mathbb{R}^{c\times h\times w}$ is the squeeze-and-excitation operation. It learns a channel-wise rescaling

$$s(\mathbf{u})_a = h(g(\mathbf{u}))_a \, \mathbf{u}_a \tag{3.3}$$

of each channel $a \in [\![c]\!]$ by first aggregating spatial information and then computing scaling factors for each channel. The spatial aggregation is performed by $g : \mathbb{R}^{c\times h\times w} \to \mathbb{R}^c$ which is implemented through global average-pooling, i.e. taking the average of each channel $a \in [\![c]\!]$ as follows:

$$g(\mathbf{x})_a = \frac{1}{hw}\sum_{i=1}^{h}\sum_{j=1}^{w}\mathbf{x}_{a,i,j}. \tag{3.4}$$

The resulting vector is fed into a subnetwork $h : \mathbb{R}^c \to \mathbb{R}^c$. It consists of two sequential dense layers with a ReLU non-linearity in between. The output is a vector of scaling factors

$$h(\mathbf{x}) = \sigma\left(\mathbf{W}_2\mathrm{ReLU}\left(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2\right), \tag{3.5}$$

where σ is the sigmoid function to keep outputs between zero and one. This way, the factors can be used to scale each channel of the original input feature map, effectively re-weighting the importance of features. More details on the SE-ResNeXt50 architecture can be found in [43].

3.4 Experiments

The previously described network architecture is used to classify the C-NMC dataset. Since the dataset is imbalanced, the overall accuracy can be misleading, and we report accuracy, sensitivity, and specificity, as well as weighted F1-scores, weighted precision, and weighted recall. For sensitivity and specificity, the ALL-class is regarded as the positive class. Furthermore, because the data is from multiple subjects, we report subject-level accuracies.

Hyperparameters for the training procedure have been chosen by validating on the

preliminary test set. The chosen hyperparameters are used for multiple training runs with different random seeds, and the best model according to the F1-score on the preliminary test set is selected for our competition entry. Final results on the test set are taken from the online challenge leaderboard.

3.4.1 Data Augmentation

Even though the dataset contains more than 10,000 images, several data augmentation techniques can be applied to increase the amount of training data further and improve the training of our convolutional neural network.

Since microscopic images are invariant to flips and rotations, we perform horizontal and vertical flips with 50 % probability each and rotations with an angle chosen uniformly at random from $[0, 360)$ degrees. Since convolutional neural networks with pooling operations or strides larger than one are not perfectly translation invariant, we also perform random translations of up to 20 % of each side-length in horizontal and vertical directions.

We do not randomly scale the images, because cell size may be a diagnostic factor to differentiate between ALL and normal cells [15]. Furthermore, we do not apply any brightness or color augmentation due to this dataset's stain-normalization preprocessing. Both data augmentation methods are commonly used but would lead to an unnecessary distribution shift between training and test set on this specific dataset.

Additionally, the images are center-cropped to 300×300 pixels to decrease the dimensionality of the input data. This will generally make learning a classifier faster and easier. Even though the cropping discards large parts of the image, it has no effect on the classification accuracy, because only very few cells are actually larger than this crop. In many cases, images that are not completely black outside of the crop are segmentation failures that include parts of the background.

3.4.2 Training and Testing

The network is pre-trained on ImageNet and then fine-tuned on the C-NMC training set. Because there are only two classes, the network output is a single value that indicates the class with its sign. The loss function is the weighted binary cross-entropy

$$\ell_{\text{bin.xent}}\left(\widehat{y}, y\right) = -wy \log\left(\sigma\left(\widehat{y}\right)\right) - (1-y) \log\left(1 - \sigma\left(\widehat{y}\right)\right), \tag{3.6}$$

where σ is the sigmoid function, $\widehat{y} \in \mathbb{R}$ is the network output, $y \in \{0, 1\}$ is the true label, and $w = \frac{\text{negative}}{\text{positive}}$ is the ratio of negative to positive examples in the training set. The weight w helps to deal with the class imbalance present in the dataset by scaling the gradient for positive examples with a coefficient larger than one.

The network is fine-tuned for 6 epochs using Adam [59] with a batch size of 16. The learning rate is decayed using a step function that starts at $\eta_{\text{base}} = 1$ and is divided

Table 3.2: Results on the preliminary test set over 24 training runs using the best model among the checkpoints of each run, measured by F1-score.

	Minimum	Mean ± Stddev.	Maximum
Accuracy	86.40	87.96 ± 0.90	89.88
Sensitivity	88.43	92.01 ± 1.44	94.50
Specificity	75.31	80.36 ± 2.39	84.72
F1-score	86.28	87.89 ± 0.90	89.81
Precision	86.27	87.91 ± 0.90	89.81
Recall	86.40	87.96 ± 0.90	89.88

by 10 every 2 epochs. Every stage of the network, i.e. a stack of building blocks that operate on the same spatial resolution as described in Section 3.3 and [144], uses its own learning rate derived from η_{base} with a stage-specific factor. The first and second stages have an effective learning rate of $\eta_{12} = 10^{-6}\eta_{\text{base}}$, the third to fifth stages use $\eta_{345} = 10^{-4}\eta_{\text{base}}$, and the last dense layer uses $\eta_6 = 10^{-2}\eta_{\text{base}}$.

This results in lower effective learning rates in the earlier layers of the network and is desirable, because the network is initialized with weights from ImageNet pre-training. These have been shown to contain generally useful image filters in the lowest layers [148], and continuing to optimize these weights on a small dataset with large learning rates will degrade these general filters.

During inference, we present 8 rotated versions of each image and average the network output to further improve classification results.

3.4.3 Model Selection

We find that results are sensitive to the random seed, despite all networks starting from the same weight initialization due to the pre-training. Therefore, we conduct 24 training runs with different random seeds and measure the F1-score on the preliminary test set over the course of the training. The F1-score includes both precision and recall and is a better choice than the plain accuracy on this dataset, since it is insensitive to the strong class imbalance. It is also the metric that is used to judge submissions for the competition. We collect the 24 model checkpoints that have the highest F1-scores in their respective training run and report results in Table 3.2.

The best model by F1-score achieves 89.81 % on the preliminary test set. Figure 3.2 shows the evolution of loss and F1-score during its training process, while Figure 3.3 shows cell classification accuracy grouped by subject. Due to the pre-training and comparatively small target dataset, accuracy quickly stagnates. Increasing the amount of training epochs leads to worse results as the pre-trained weights are distorted more and more.

Overall, classification for healthy subjects is worse, because the specificity (true

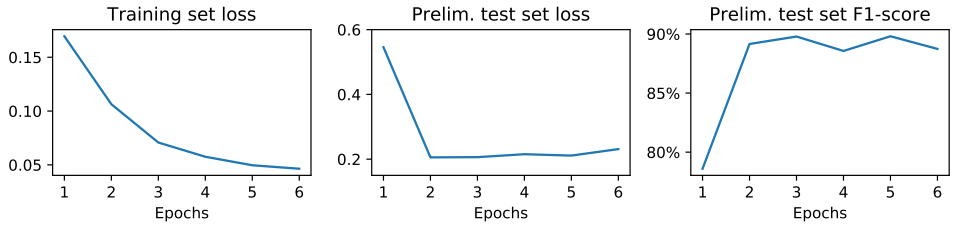

Figure 3.2: Training and validation curves of the best model after each training epoch. The model achieves the maximum F1-score after the 5th training epoch.

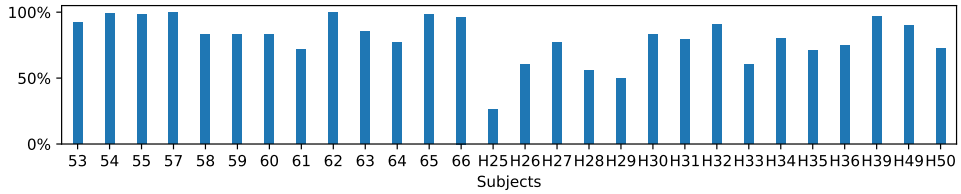

Figure 3.3: Subject-level cell classification accuracy on preliminary test set using the best model. Subjects with the prefix H are healthy.

negatives divided by total negatives in the dataset) is comparatively low. Consequently, false positives can be expected when using the network for diagnosis of patients. Still, false positives are preferable to false negatives when trying to diagnose cancer with an automated method. While such a diagnosis is stressful for the patient, it would be far worse to misdiagnose when cancer is actually present.

3.4.4 Results on the Final Test Set

We select the best model according to F1-score and use it to classify the final test set and submit the result to the online evaluation service. The model achieves a weighted F1-score of 88.91 % and third place on the final leaderboard.

3.4.5 Ablation Studies

Our main design choices for the training and testing procedures are layer-specific learning rates and test-time rotations. In order to show their impact we perform two ablation studies:

- *NOROT*: The training procedure is unchanged, but during testing the images are only presented in their original orientation.

- *NOSPECLR*: During training, every layer has an effective learning rate of $\eta_{\text{all}}\eta_{\text{base}}$ while still using the scheduled decay for η_{base} as described previously. We test $\eta_{\text{all}} \in \{10^{-3}, 10^{-4}, 10^{-5}\}$ and report results for the best setting $\eta_{\text{all}} = 10^{-4}$. The testing procedure is unchanged.

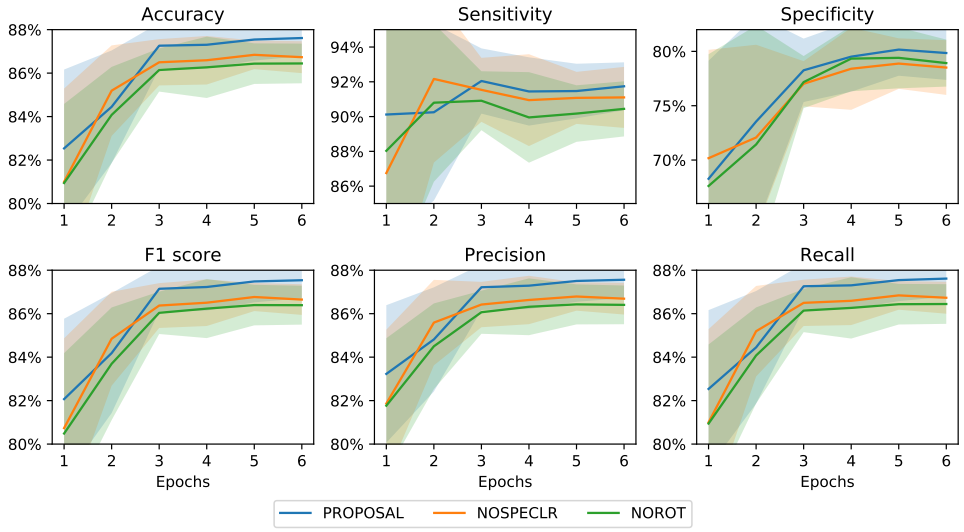

Figure 3.4: Metrics on the preliminary test set measured after each training epoch for the proposed setting and the two ablation studies. Lines display the mean over 24 training runs and the shaded area marks one standard deviation.

Figure 3.4 shows results on the preliminary test set for 24 repetitions of our proposed setting and the two ablation studies. As expected, both ablations decrease performance in terms of all considered metrics. Not using test-time rotations significantly ($p < 0.001$, one-sided Mann-Whitney U test) decreases the mean F1-score from $87.89 \pm 0.90\,\%$ to $86.92 \pm 0.81\,\%$. The influence of layer-wise learning rates is significant ($p < 0.002$) but smaller: A constant learning rate for all layers decreases the mean F1-score to $87.20 \pm 0.70\,\%$. In summary, both the test-time rotations and layer specific learning rates lead to a statistically significant improvement.

3.5 Conclusion

We present a simple, yet effective method to automatically classify white blood cell microscopic images into normal B-lymphoid precursors and malignant B-lymphoblasts. It is powered by a ResNeXt convolutional neural network with squeeze-and-excitation modules. Even though the chosen network is large and there is little training data, good results can be achieved with pre-training and carefully tuned hyperparameters. While pre-training achieves goal (G3) by allowing training with small datasets, the system becomes even more sensitive to hyperparameters such as the learning rate and the number of training iterations. This further highlights the need for automatic hyperparameter search methods. We observe significant variance in final performance due to non-determinism in the training procedure. This will also be an issue in later chapters that deal with NAS, because these noisy performance measurements make it difficult to compare architectures fairly.

Part II

Evolutionary Weight Optimization

Evolutionary algorithms are the second important concept next to deep neural networks that this thesis makes extensive use of. Chapter 4 introduces EAs that use a population of solutions to solve black-box optimization problems. The introduced algorithm is then showcased with applications to weight training for DNNs in the context of supervised learning and reinforcement learning in Chapter 5 and 6 respectively. This exploration of EAs in the context of DNN training serves to answer the question if large DNNs with fixed architectures can be trained by EAs and achieve competitive results to SGD. This would be a requirement for successful neuroevolution methods for large-scale DNNs that do not use SGD.

Chapter 4

Population-based Evolutionary Algorithms

During the sixties, the ideas that are now subsumed under the term evolutionary algorithm are developed independently at first [52]. Evolution strategies [99, 112] are developed by Rechenberg and Schwefel as a method for numerical optimization, evolutionary programming [27] is developed by Fogel et al. as a method to automatically design agents powered by finite state machines, and the genetic algorithm [40] is developed by Holland as a general adaptive system. Today, there exists a wealth of related algorithms that stem from these roots. All have in common that they are inspired by evolution in nature but do not try to faithfully reproduce it. Instead, the goal is to create reduced computational models that have all components that are necessary for evolution to progress.

While evolutionary algorithms are not exclusively used for optimization, viewing them as a class of nature-inspired black-box optimization algorithms is common. In this chapter, we focus on evolutionary algorithms that maintain a set of solutions, called a population, and continually improve it in a process inspired by the evolutionary process in nature. This is in contrast to natural evolution strategies, which work with probability distributions to represent solutions instead of a population and are quite far removed from any natural analogies. They will be introduced later in Section 8.3.

In contrast to numerical optimization methods stemming from mathematics, such as line search, Newton's method, gradient descent, etc. that work with real-valued solutions, EAs are additionally applicable to more exotic solution spaces like graphs. This makes them attractive for problems like architecture optimization that have a natural graph representation.

Outline. First, Section 4.1 introduces the notion of a stochastic black-box optimization problem and how it relates to the problems discussed in this thesis. Then, Section 4.2 gives a birds-eye view of a population-based generational EA. The following sections describe different parts of the EA in more detail. Section 4.3 deals with representation of solutions, Section 4.4 deals with variation operators, and finally Section 4.5 describes the selection process.

4.1 Stochastic Black-Box Optimization

Consider an optimization problem $\max_x u(x)$ with $x \in \mathcal{X}$ over an arbitrary solution space $\mathcal{X}$. The function $u(x)$ measures a solution's fitness, and we are searching for the solution x^* with the maximum fitness. The problem is called black-box if $u(x)$ is not given in closed form and can only be evaluated at arbitrary points x. EAs work under this black-box assumption, which makes them applicable to basically any optimization problem. It follows from the black-box assumption that we cannot differentiate $u(x)$ w.r.t. x, which makes it impossible to apply gradient-based optimization methods unless a numerical approximation of the gradient is feasible, e.g. in very low dimensional problems. Even without the black-box assumption, non-differentiable fitness functions often arise when the function depends on some kind of simulation or there are discrete choices encoded in x that influence the fitness.

Depending on the problem, the fitness function might be stochastic, i.e. $u(x)$ is a probability distribution over fitness values conditioned on the solution x. Optimizing for the expected value $\mathbb{E}[u(x)]$ is usually desirable, but we cannot determine it exactly due to the problem's black-box nature. It becomes necessary to sample from the distribution and calculate the average value to get an approximation of the expected value. However, calling the fitness function is expensive in many cases, which is why an algorithm that uses as few fitness function evaluations as possible is preferred. Such an optimization in the presence of fitness noise stemming from the approximation from samples is a difficult problem that has long been discussed in literature. EAs in particular are well-equipped to deal with noise, because a diverse population of solutions is maintained [4, 11]. Even if some solutions are discarded because of apparent low fitness despite being a good solution, this is unlikely to happen to all good solutions in the population at once.

We will later use EAs to optimize DNN weights and architectures. In the latter problem setting, we will find discrete choices that make the fitness function non-differentiable and make it an ideal candidate for the application of an EA. Furthermore, for both weight and architecture optimization, we will have to deal with stochastic fitness functions. During weight optimization in Chapter 5, the fitness function is noisy because fitness evaluation will use small sampled batches of data instead of the full dataset. During architecture optimization in Chapter 7, the fitness noise stems from the non-deterministic SGD training procedure for DNNs.

4.2 Evolutionary Algorithm

Population-based generational EAs solve (possibly black-box and stochastic) optimization problems by creating a population of solutions and refining it over multiple generations to increase the average fitness. Compared to optimizing a single solution, it might seem wasteful to optimize a whole population of solutions if, in the end,

the single best result is sufficient and the rest can be discarded. However, such an approach prevents premature convergence to good local solutions that could trap the optimization, and it makes the optimization process more resilient to fitness noise. Furthermore, in some problem settings having a population of good solutions can be desired. If we consider the optimization of DNNs using an EA, the population at the end of the optimization consists of several well-performing but different models which make promising candidates for ensembling. This ensemble gets naturally created for free as a byproduct of the optimization.

The process of refining the population of solutions is inspired by the evolutionary process that happens in nature. Algorithm 1 contains pseudo-code for a population-based generational EA. After initializing the population, a simple loop of variation, evaluation, and selection is repeated for a number of generations or until the solution is satisfactory. All these steps are problem-dependent and are described in more detail in the following sections.

Since Algorithm 1 gives only a birds-eye view of an EA, we cannot yet see all hyperparameters that are in play and control the algorithm. Next to the population size μ and offspring count λ, the variation operators have hyperparameters themselves, which depend on their implementation. Most notably, mutation is usually controlled by a mutations strength that controls how close (on average) a mutated offspring solution is to its parent. Such hyperparameters can be static or dynamic, and we will see this in more detail when we apply EAs to DNN weight training in Chapter 5.

Algorithm 1: High-level view of a population-based generational $(\mu + \lambda)$ evolutionary algorithm.

```
P ← randomly initialize μ solutions x_1, x_2, ..., x_μ
evaluate fitness u(x) for all x ∈ P
while termination condition not met do
    Q ← create λ offspring by variation
    evaluate fitness u(x) for all x ∈ Q
    P ← select μ best solutions from P ∪ Q
end
```

4.3 Representation of Solutions

Depending on the problem to be optimized, different representations for solutions may be possible. The representation is chosen in conjunction with the variation operators that need to operate on this representation. For example, the well-known traveling salesman problem has been approached with EAs using many different representations like binary indicators, paths as strings of identifiers, adjacency matrices, etc. all with their matching variation operators [44]. It is desirable to find a representation that

increases the likelihood of good offspring being produced through the application of the variation operator. In other words, different representations admit different variation operators, and both should be chosen so as to minimize the chance of offspring with low fitness being produced from parents with high fitness.

In general, a distinction between genotype and phenotype of a solution can be made. The genotype would be the representation that the variation operators work with, while the phenotype is derived from the genotype and used for the evaluation. Such an indirect encoding is employed by HyperNEAT [121] to encode the weights of a neural network. The genotype here actually contains a compositional pattern producing network (CPPN) [119], which is evolved by the NEAT [123] algorithm. A CPPN is similar to a neural network in that it is a graph with weighted edges and its nodes apply an activation function to the sum of its inputs. However, each node has its own activation function and the topology is not layered but evolved arbitrarily. This design allows to create CPPNs whose outputs exhibit typical patterns found in evolution, like repetition or symmetries. In HyperNEAT, the genotype CPPN is then sampled at different positions to create weights for another neural network, which is the phenotype in this scenario.

However, in many cases a direct encoding where genotype and phenotype coincide can be a good choice as well. There are numerous examples of EAs being applied to real-valued vectors in numerical optimization.

4.4 Variation Operators

The implementation of variation operators is problem-dependent and also needs to be designed to fit the chosen genotype representation. A variation operator creates new solutions from solutions that currently exist in the population by performing either a recombination of multiple parent solutions, a mutation of a parent solution, or both. Depending on the representation and structure of the solution space, some variation operators would be more likely to produce offspring with high fitness, and they should be preferred.

There are two main kinds of variation operators called crossover and mutation. A crossover operator is inspired by the sexual reproduction of animals and combines two or more solutions into an offspring solution. The goal is to copy parts of a well-performing parent solution into an offspring solution to combine features from multiple phenotypes. A mutation operator, on the other hand, is only applied to a single solution and slightly modifies it. This is again in similarity to random gene mutations happening during reproduction in nature and is expected to produce mostly small incremental changes in the phenotype.

When the genotypes are real-valued vectors, there are a number of standard crossover and mutation operators that can be applied. The uniform crossover of

two parents $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^d$ creates offspring $\mathbf{z} \in \mathbb{R}^d$ as

$$\mathbf{z}_i = \begin{cases} (\mathbf{x}_1)_i & \text{with probability } 50\,\% \\ (\mathbf{x}_2)_i & \text{else} \end{cases} \tag{4.1}$$

by randomly deciding which elements $i \in [\![d]\!]$ of the offspring's solution vector is copied from which parent. While this randomized crossover operator works well in many cases, if solutions are arranged in such a way that linear combinations of their representations are meaningful, the arithmetic crossover can be a better choice. Arithmetic crossover creates offspring

$$\mathbf{z} = \frac{1}{2}\mathbf{x}_1 + \frac{1}{2}\mathbf{x}_2 \tag{4.2}$$

from two parents $\mathbf{x}_1$ and $\mathbf{x}_2$ by taking the arithmetic mean. In principle, this weighting does not need to be equal and could, for example, be biased so that the parent with higher fitness has a higher coefficient than the other. Both crossover operators can be extended to more than two parents.

The most common mutation operator for real-valued vectors is the random normal mutation. This mutation operator creates offspring

$$\mathbf{z} = \mathbf{x}_1 + \sigma\epsilon \tag{4.3}$$

by adding random noise ϵ drawn from a normal distribution $\mathcal{N}(0, 1)$ and scaled by a mutation strength $\sigma \in \mathbb{R}^+$ to a parent $\mathbf{x}_1$. Under this mutation, most components of the solution will change only slightly, whereas a few components may change a lot. If other behaviors are desired, different probability distributions can be employed.

How exactly crossover and mutation are combined is a matter of preference and many different variants exist. For example, crossover might be used to create a certain percentage of the offspring, and mutation might be used to create the remaining offspring solutions. Alternatively, offspring created by crossover might also be affected by mutation, or only a certain percentage of offspring might be affected by mutation. There is a lot of freedom in designing the variation operation of an EA as these choices are very problem dependent.

4.5 Selection

Only applying variation operators to random solutions from the population is similar to performing a random search in the vicinity of the solutions covered by the population. In order to drive the population to higher average fitness, selection operators are needed. They decide which solutions survive a generation and become eligible as parents for the next generation. This process applies selective pressure to the population and, over time, eliminates weak solutions so that the average fitness of the population

increases. This in turn makes it more likely that the variation operators are applied to good solutions and are more likely to produce good offspring solutions.

The selection operator is unrelated to the representation of solutions, and therefore the same set of selection operators is applicable to all problems. Still, an appropriate choice must be made depending on the problem that is optimized. They vary in amount of selective pressure, i.e. how strongly better solutions are preferred over worse solutions. The choice of selection operator will therefore decide how the fitness distribution changes from generation to generation. While it might seem to be always desirable to select only a few of the best solutions, a very high selective pressure will reduce diversity in the population and can therefore counteract the advantages of a population-based algorithm. Care has to be taken to strike a good balance between exploitation of good solutions and exploration of different solutions.

In this thesis, we will always work with truncation selection. This selection scheme simply sorts the offspring population and then selects the μ best solutions as the next generation's population. As long as μ is large enough, this approach still selects a diverse set of solutions. Whether the offspring population that we select from is the union of the previous generation's parents and their offspring $P \cup Q$ or just their offspring Q is a design choice. The former choice is called plus-selection, while the latter choice is called comma-selection. It is common to speak of $\left(\mu \stackrel{+}{,} \lambda\right)$ EAs to show which kind of selection process is used.

Comma-selection prevents solutions from staying in the population unchanged for more than one generation. This can be helpful if the fitness function is stochastic, because in such a situation a solution may receive a very high fitness value by chance even though the solution is bad. Using plus-selection, this solution would be kept in the population and used to produce offspring until even better individuals are found. This can be very detrimental and is avoided by comma-selection.

One disadvantage of comma-selection is that legitimately good solutions only get a single chance to produce offspring as well and are then discarded. Especially with fitness functions that are expensive to evaluate this can be undesirable. Elite selection can be used in conjunction with comma-selection to prevent this. Elite selection simply copies a fixed number of the best solutions from the last generation into the next generation and effectively bypasses the actual selection scheme for these elite solutions.

Chapter 5

Application to Deep Supervised Learning

Stochastic gradient descent is the leading approach for neural network weight optimization. Significant research effort has lead to creations such as the Adam optimizer [59], batch normalization layers [47], or advantageous weight initializations [30], all of which improve upon the standard gradient-based training process. Furthermore, efficient libraries with automatic differentiation and GPU support are readily available. It is therefore unsurprising that SGD outperforms all other approaches to neural network training. Still, in this chapter we want to examine EAs for this task because of their prominent advantage of being black-box, i.e. not needing gradient information.

While neural networks are usually built so that they are differentiable, this restriction can be lifted when training with EAs. For example, this would allow the direct training of neural networks with binary weights for deployment in low-power embedded devices. As a second example, the loss function does not need to be differentiable when training with an EA, so that it becomes possible to optimize for different metrics like the F1-score directly. We, however, are mostly motivated by the prospect of being able to optimize weights and architecture of a DNN together in a unified evolutionary algorithm. While earlier work on neuroevolution sparked algorithms like NEAT that promise exactly that, these approaches only work well for small networks on the order of tens of nodes [122, 124, 140]. NEAT and other neuroevolution approaches do not scale to large DNNs and the large datasets that are common today in the field of deep learning. On the other hand, evolutionary approaches different from neuroevolution are attracting attention for DNN training [85, 6, 152] but often study only small-scale DNNs or problems that are not representative of today's large network architectures and datasets. Therefore, we want to scale up an evolutionary approach and evaluate its usefulness for large-scale DNN training, which we consider a prerequisite for its successful application in neuroevolution methods.

With growing computational resources and algorithmic advances, it is becoming feasible to optimize large, directly encoded neural networks with EAs. The limited evaluation evolutionary algorithm (LEEA) [85] saves computation by performing the fitness evaluation on small batches of data and smoothing the resulting fitness

noise with a fitness inheritance scheme. We create a LEEA implementation that executes entirely on a GPU to facilitate extensive experimentation with larger DNNs and populations than previously possible. The GPU implementation avoids memory bandwidth bottlenecks, reduces latency, and most importantly allows to efficiently batch the evaluation of multiple network instances with different weights into a single operation.

Using this framework, we perform experiments with significantly larger networks and on more difficult problems than in the original publication [85]. We highlight a trade-off between batch size and achievable accuracy and also find the proposed fitness inheritance scheme to be detrimental. Instead, we show how the LEEA can profit from low selective pressure when using small batch sizes. Despite the problems discussed in literature about crossover and neural networks [28, 131], we see that basic uniform and arithmetic crossover perform well when paired with an appropriately tuned mutation operator. Finally, we apply the lessons learned to train a neural network with 92,000 parameters on MNIST using an EA and achieve 97.6 % test accuracy. In comparison, training with Adam results in 98 % test accuracy. Note that these low accuracies (for MNIST) are due to the network architecture and downsampled inputs and therefore neither the EA nor Adam would be able to reach state-of-the-art results.

The results presented in this chapter are based on the following publication with accompanying source code:

- Jonas Prellberg and Oliver Kramer. Limited evaluation evolutionary optimization of large neural networks. In Frank Trollmann and Anni-Yasmin Turhan, editors, *KI 2018: Advances in Artificial Intelligence - 41st German Conference on AI, Berlin, Germany, September 24-28, 2018, Proceedings*, volume 11117 of *Lecture Notes in Computer Science*, pages 270–283. Springer, 2018

- `https://github.com/jprellberg/gpuea`

Outline. Section 5.1 presents related work on the application of EAs to neural network training. In Section 5.2, we present our GPU-accelerated EA for the evolution of DNN weights. Section 5.3 covers all experiments and contains the main results of this work. Finally, we conclude the chapter with a summary in Section 5.4.

5.1 Evolutionary DNN Weight Optimization

The limited evaluation evolutionary algorithm for neural network training by Morse et al. [85] is a modified population-based generational EA that picks a small batch of training examples at the beginning of every generation and uses it to evaluate the population of neural networks. This idea is conceptually very similar to SGD, which also uses a batch of data for each step. Performing the fitness evaluation on small batches instead of the complete training set significantly reduces the required

computation, but it also introduces noise into the fitness evaluation. The second component of the LEEA is therefore a fitness inheritance scheme that combines an offspring solution's fitness evaluation result with its parent's fitness. The algorithm is tested with networks of up to 1,500 weights and achieves results comparable to SGD on small datasets.

Baioletti et al. [6] pick up the idea of limited evaluation but replace the evolutionary algorithm with differential evolution, which is a very successful optimizer for continuous parameter spaces [22]. The largest network they experiment with contains 7,000 weights. However, there is still a rather large performance gap on the MNIST dataset between their best performing differential evolution algorithm at 85 % accuracy and a standard SGD training at 92 % accuracy.

Yaman et al. [147] combine the concepts limited evaluation, differential evolution, and cooperative co-evolution. They consider the weights of a single node's inputs as a component and evolve many populations of such components in parallel. Complete solutions are created by assembling components from different populations into a complete network. Using this approach, they are able to optimize networks of up to 28,000 weights.

Zhang et al. [152] explore neural network training with a natural evolution strategy. This algorithm starts with a parent weight vector and creates many so-called pseudo-offspring weight vectors by adding random noise to it. The fitness of all pseudo-offspring is evaluated and used to estimate the gradient in the direction of lower loss. Finally, this gradient approximation is fed to SGD or another optimizer, such as Adam, to modify the parent weight vector. Using this approach, they achieve 99 % accuracy on MNIST with 50,000 pseudo-offspring for the gradient approximation. Natural evolution strategies will be explained in more detail in Section 8.3.

5.2 Accelerating Evolutionary DNN Weight Optimization with GPUs

We implement a population-based EA that optimizes the weights of directly encoded, fixed size neural networks. For performance reasons, the EA is implemented with TensorFlow and executes entirely on the GPU, i.e. the whole population of networks lives in GPU memory and all EA logic is performed on the GPU.

5.2.1 Evolutionary Algorithm

Algorithm 2 shows our EA in pseudo-code. It is a population-based (μ, λ) EA extended by the limited evaluation concept. In our case, we set $\mu = \lambda$ so that the initial population contains λ solutions, which are randomly initialized neural network weight vectors. Every generation, the fitness evaluation of neural networks in the population P is performed on a small batch of data $\mathbf{x}, \mathbf{y}$ that is drawn randomly from

Algorithm 2: Evolutionary algorithm for supervised DNN training.

Let $\mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta)$ be the loss for a batch of data $\mathbf{x}, \mathbf{y}$ under weights θ
$P \leftarrow$ randomly initialize λ weight vectors $\theta_1, \theta_2, \ldots, \theta_\lambda$
while termination condition not met **do**
  $\mathbf{x}, \mathbf{y} \leftarrow$ get random batch of training data
  evaluate fitness $-\mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta)$ for all $\theta \in P$
  $T \leftarrow$ select $\rho\lambda$ best solutions from P by truncation selection
  $E \leftarrow$ select $p_E\lambda$ best solutions from P as elites
  $C \leftarrow$ apply crossover operator to $p_C\lambda$ random parent pairs from $T \times T$
  $M \leftarrow$ apply mutation operator to $p_M\lambda$ random parents from T
  $P \leftarrow E \cup C \cup M$
end

the training set. During fitness evaluation, every solution $\theta \in P$ is assigned a fitness value

$$u_{\mathbf{x},\mathbf{y}}(\theta) = -\mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta), \tag{5.1}$$

which is the negative of the (problem-dependent) loss function $\mathcal{L}(\theta)$ restricted to data and labels $\mathbf{x}, \mathbf{y}$ instead of the whole dataset. By convention, fitness values are maximized, which is why in our case the fitness is the negative of the loss function, which has to be minimized. Therefore, the best solution is the one with highest fitness and lowest loss. The restriction to data and labels from $\mathbf{x}, \mathbf{y}$ reduces the computational cost, because the networks have to be evaluated on less data, but it also introduces an increasing amount of fitness noise, especially with smaller batch sizes. To counteract this, Morse et al. [85] propose a fitness inheritance scheme that we implement as well.

Due to the stochasticity in the fitness evaluation, it seems advantageous to combine fitness evaluation results from multiple batches. However, simply evaluating every network on multiple batches is no different from using a larger batch size and would cancel out any performance gains. Therefore, the assumption is made that the fitness of a parent network and its offspring are related. Then, a parent's fitness can be inherited to its offspring as a good initial guess and be refined by the actual fitness evaluation of the offspring. This is done in form of a weighted sum

$$u_{\text{adj}} = (1 - \alpha)\, u_{\text{inh}} + \alpha u_{\mathbf{x},\mathbf{y}}(\theta), \tag{5.2}$$

where u_{inh} is the fitness value inherited by the parents, $u_{\mathbf{x},\mathbf{y}}(\theta)$ is the fitness value of the offspring θ on the current batch $\mathbf{x}, \mathbf{y}$, and $\alpha \in [0, 1]$ is a hyperparameter that controls the strength of the fitness inheritance scheme. Setting α to one disables fitness inheritance altogether. During crossover of two parents with fitness u_1 and u_2 or during mutation of a single parent with fitness u_3, the inherited fitness values are $u_{\text{inh}} = \frac{1}{2}(u_1 + u_2)$ or $u_{\text{inh}} = u_3$ respectively.

After the fitness evaluation is finished, truncation selection is applied to create a set of parents T that are eligible for reproduction. The size of this set is controlled by the selection proportion $\rho \in [0, 1]$. The smaller it is, the higher the selective pressure, since only the best solutions will become parents for the next generation.

Then, a total of λ offspring networks are derived from T. The hyperparameters p_E, p_C, and p_M determine the percentage of offspring created by elite selection, crossover, and mutation respectively and must sum to one. First, the $p_E\lambda$ networks with the highest fitness are selected as elites from the population. These elites move into the next generation unchanged and will be evaluated again. Even though their weights did not change, the repeated evaluation is desirable. Because the fitness function is only evaluated on a small batch of data, it is stochastic and repeated evaluations will result in a better estimate of the true fitness when combined with previous fitness evaluation results. Even if the fitness inheritance is turned off, this behavior is still desirable so that all fitness values in a generation are based on the same random batch of data and can be compared fairly.

Next, $p_C\lambda$ pairs of networks are selected uniformly at random from T as parents for crossover, and finally $p_M\lambda$ networks are selected uniformly at random from T as parents for mutation. By applying the crossover and mutation operators described in the next section, this results in $p_C\lambda$ and $p_M\lambda$ new offspring networks. Together with the elites, the offspring population has λ members and replaces the current population.

5.2.2 Crossover and Mutation Operators

Members of the EA population are direct encodings of neural network weights $\theta \in \mathbb{R}^d$, where d is the total number of weights in each network. The crossover and mutation operators directly modify this vector representation. Therefore, we can use the general variation operators for real-valued vectors described in Section 4.4. Depending on the experiment, we employ either random uniform crossover or arithmetic crossover, and random normal mutation with mutation strength $\sigma \in \mathbb{R}^+$.

The mutation strength σ is an important hyperparameter that can be changed over the course of the EA run if desired. In the simplest case, the mutation strength stays constant over all generations, but different mutation strengths may be necessary, depending on the chosen crossover operator. Therefore we also experiment with two different mutation strength adaptation schemes.

First, we implement deterministic control in the form of an exponentially decaying value. For each generation i, the mutation strength is calculated according to

$$\sigma_i = 0.99^{\frac{i}{k}}\sigma_0, \tag{5.3}$$

where σ_0 is the initial mutation strength and the hyperparameter k controls the decay rate in terms of generations.

Furthermore, we implement self-adaptive control. The mutation strength σ is

included as a gene in each solution's genotype, and each solution is mutated with the mutation strength σ taken from its own genotype. The mutation strength itself is mutated according to

$$\sigma_{i+1} = \sigma_i \exp(\tau\epsilon) \tag{5.4}$$

with ϵ drawn from a normal distribution $\mathcal{N}(0, 1)$ and hyperparameter $\tau \in \mathbb{R}^+$. During crossover, the arithmetic mean of two σ-genes produces the value for the σ-gene in the offspring.

5.2.3 GPU Implementation

Naively executing thousands of small neural networks on a GPU in parallel incurs significant overhead, since many short-running, parallel operations that compete for resources are launched, each of which also has a startup cost. To efficiently evaluate thousands of network weight configurations, the computations should be expressed as batch tensor products where possible.

Consider a neural network containing a dense layer as described in Section 2.3.1. We assume inputs are of dimensionality d_1, and the dense layer has d_2 output units and no bias. In such a case, a simple matrix-vector product between the dense layer weight $\mathbf{W} \in \mathbb{R}^{d_2 \times d_1}$ and input $\mathbf{u} \in \mathbb{R}^{d_1}$ is performed. In actual implementation, the input tensor has an additional batch dimension, and a batched matrix-vector product is computed between $\mathbf{W}$ and an input batch $\mathbf{u} \in \mathbb{R}^{b \times d_1}$ consisting of b independent samples. Every sample in the input batch is processed individually and multiplied with $\mathbf{W}$, but this is expressed as an efficient GPU operation.

Batching over multiple sets of network weights follows the same approach and introduces a population dimension with p components. Obviously, the weight tensor needs to be extended by this dimension so that it can hold weights of different networks. However, the data tensor also needs an additional population dimension, because the output of each layer will be different for networks with different weights. This will result in a product between weights $\mathbf{W} \in \mathbb{R}^{p \times d_2 \times d_1}$ and data $\mathbf{u} \in \mathbb{R}^{p \times b \times d_1}$, where the batched matrix-vector product is computed for every pair of weights and data that occupy the same component in the population dimension.

In order to exploit this batched evaluation of populations, the whole population lives in GPU memory in the required tensor format. Next to enabling the population batching, this also alleviates the need to copy data between host and GPU memory, which is typically a bottleneck. These advantages apply as long as the networks do not have too many weights. The larger each network, the more computation is necessary to evaluate it, which reduces the gain from batching multiple networks together. Furthermore, combinations of population size, network size, and batch size are limited by the available GPU memory. Despite these shortcomings, this framework allows us to experiment at reasonably large scales, such as a population of 8,000 networks with 92,000 parameters and a batch size of 64 using 16 GB GPU memory.

Table 5.1: Default hyperparameter settings for exploratory experiments on the validation set and the final experiment on the test set. The differences, informed by the results of experiments on the validation set, are marked in bold.

Parameter	Validation value	Test value
Crossover operator	uniform	uniform
Mutation strength adaptation	constant	constant
Batch size	512	**1024**
Elite ratio p_E	0.05	0.05
Crossover ratio p_C	0.50	**0.75**
Mutation ratio p_M	0.45	**0.20**
Population size λ	1000	**2000**
Mutation strength σ	0.001	0.001
Truncation proportion ρ	0.50	0.50
Fitness inheritance weight α	1.00	1.00

5.3 Experiments

We apply the EA described in the previous Section 5.2 to optimize weights $\theta \in \Theta$ of a neural network $f(\theta, \mathbf{x}) : \Theta \times \mathcal{X} \to \mathcal{Y}$ with $\mathcal{X} = \mathbb{R}^{1\times 28\times 28}$ and $\mathcal{Y} = \mathbb{R}^C$ that classifies the MNIST dataset. MNIST is a standard image classification benchmark with 28×28 pixel grayscale inputs and $C = 10$ classes. The training set contains 50,000 images, which we split into an actual training set of 45,000 images and a validation set of 5,000 images.

The fitness function that is optimized by the EA is defined over a batch of B training examples with data $\mathbf{x}^{(i)} \in \mathcal{X}$ and labels $y^{(i)} \in [\![C]\!]$, where $i \in [\![B]\!]$. We define it as the negative of the loss function (see 5.1), which in this case is the cross-entropy loss

$$\mathcal{L}_{\mathbf{x},y}(\theta) = \frac{1}{B}\sum_{i=1}^{B} \ell_{\text{xent}}\left(\widehat{\mathbf{y}}^{(i)}, y^{(i)}\right), \tag{5.5}$$

where we write $\widehat{\mathbf{y}}^{(i)} = f\left(\theta, \mathbf{x}^{(i)}\right)$ for notational convenience. The cross-entropy term ℓ_{xent} is defined in Equation 2.7.

All reported accuracies during experiments are validation set accuracies. The test set of 10,000 images is only used in the final experiment that compares the EA to SGD. All experiments have been repeated 15 times with different random seeds. When significance levels are mentioned, they have been obtained by performing a one-sided Mann-Whitney U test between the samples of each experiment. Unless otherwise stated, the hyperparameters listed in Table 5.1 are used for all experiments.

5.3.1 Network Architecture

The neural network we use in all our experiments applies 2×2 max-pooling to its inputs, followed by four dense layers with 256, 128, 64, and 10 outputs respectively.

Each dense layer except for the last one is followed by a ReLU non-linearity. In total, this network has 92,000 weights that need to be trained.

Even with SGD training this network architecture is unable to achieve state-of-the-art results, but it has been chosen due to the following considerations. We want to limit the maximum network parameter count to roughly 100,000 so that it remains possible to experiment with large populations and batch sizes. However, we also want to work with a multi-layer network. We deem this aspect important, as there should be additional difficulty in optimizing deeper networks with more interactions between weights. This is the internal covariate shift problem already described in Section 2.3.5. To avoid concentrating a large part of the weights in the network's first layer, we downsample the input. This way, it is possible to have a multi-layer network with a significant number of weights in all layers.

Furthermore, we decide against using convolutional layers, because our batched implementation of dense layers is more efficient than the convolutional counterpart. This would prolong the runtime of all experiments without providing additional insight, as we are not interested in state-of-the-art results but want to investigate whether training DNNs with an EA scales well and can perform comparably to SGD.

All networks in the EA population are initialized using the Glorot-uniform [30] initialization scheme. Even though Glorot-uniform and other neural network initialization schemes were devised to improve SGD performance, we find that the EA also benefits from them. Furthermore, this allows for a comparison against SGD on even footing.

5.3.2 Tradeoff between Batch Size and Accuracy

The EA randomly draws a batch of training data for each generation and uses it to evaluate the population's fitness. A single fitness evaluation is therefore only a noisy estimate of the true fitness that could be computed on the whole training set. The smaller the batch size, the higher the variance of this estimate, because Equation 5.5 averages over fewer cross-entropy loss values. A noisy fitness estimate introduces two problems: A good network may receive a low fitness value and be eliminated during selection, or a bad network may receive a high fitness value, survive, and reproduce.

Morse et al. [85] introduce their fitness inheritance scheme with the intent to counteract this noise and allow effective optimization despite noisy fitness values. However, in our preliminary experiments fitness inheritance does not seem to have a positive impact on training results and Morse et al. do not report experiments isolating the effect of their fitness inheritance scheme. Therefore, we perform a systematic experiment to explore the interaction between batch size, fitness inheritance weight, and the resulting network accuracy. The results can be found in Figure 5.1, and three key observations can be made.

First of all, the validation set accuracy is positively correlated with the batch size.

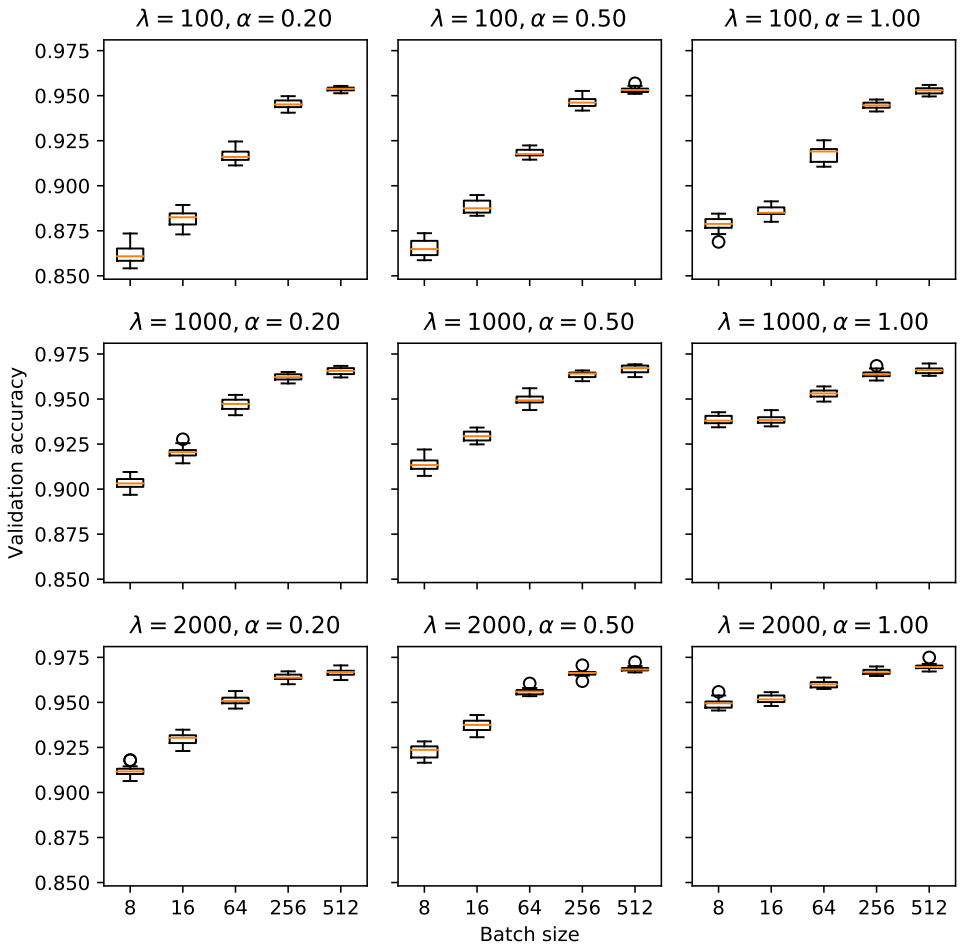

Figure 5.1: Validation accuracies of 15 EA runs for different population sizes λ, fitness inheritance strengths α and batch sizes. Looking at the grid of figures, λ increases from top to bottom, while α increases from left to right. Inside each sub-plot and for each batch-size, a boxplot depicts the lower to upper quartile values of validation accuracies achieved over the 15 repetitions, with a line at the median and whiskers that show the total range of the data.

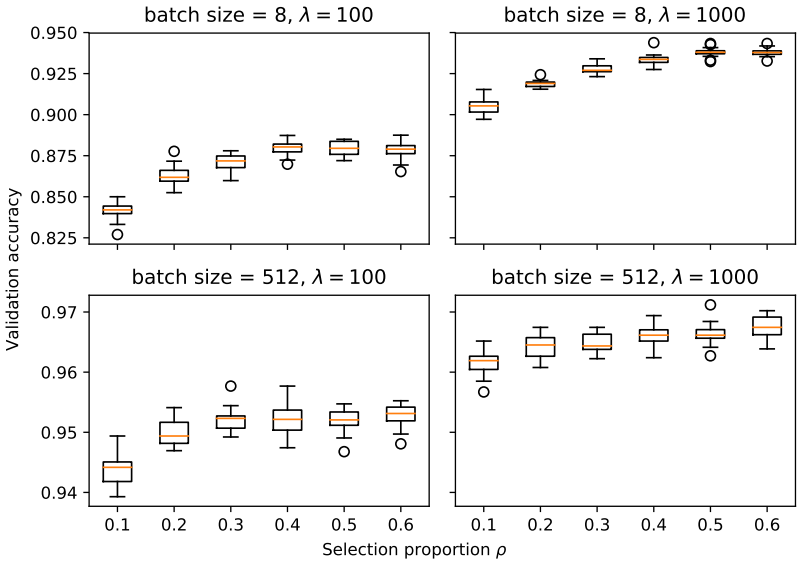

Figure 5.2: Validation accuracies of 15 EA runs for different population sizes λ, batch sizes and selection proportions ρ. The first row of figures shows results for small batch sizes, while the second row shows results for large batch sizes.

This relationship holds for all tested settings of population size λ and fitness inheritance weight α. This means, using as large a batch size as possible is always preferable. Note that the EA was allowed to run for more generations when the batch size was small, so that all runs could converge. In consequence, it is not possible to compensate the accuracy loss incurred by small batch sizes by allowing the EA to run longer.

Second, the validation set accuracy is also positively correlated with the fitness inheritance weight α. Especially for small batch sizes, significant increases in validation accuracy can be observed when increasing α. This is surprising, as higher values of α reduce the amount of fitness inheritance, which was supposed to be especially helpful for small batch sizes. Instead, we find that the fitness inheritance either has a harmful effect or no effect on validation accuracy.

Lastly, increasing the population size λ improves the validation accuracy. This is important but unsurprising, as increasing the population size is a known way to counteract noise [11]. A large population is generally a desirable property for EAs, which however comes with the downside of increased computational cost. In our case, an additional constraint is the amount of GPU memory consumed by each network, which effectively limits λ given some network size.

5.3.3 Selective Pressure

Having observed that fitness inheritance does not improve results at small batch sizes, we will now show that instead decreasing the selective pressure results in higher validation accuracies. The selective pressure influences to what degree fitter

Table 5.2: Relative improvement of validation accuracy when increasing the selection proportion from $\rho = 0.1$ to $\rho = 0.2$ in four different scenarios. Since large population sizes are also an effective countermeasure against noise, the relative improvement decreases with increasing population sizes.

Batch size	Population size	Relative improvement
8	100	2.26 %
8	1000	1.57 %
512	100	0.49 %
512	1000	0.34 %

individuals are favored over less fit individuals during the selection process. Since small batches produce noisy fitness evaluations, a low selective pressure should be helpful, because the EA is less likely to eliminate all good solutions based on inaccurate fitness estimates. In our algorithm, the selective pressure is determined by the selection proportion ρ.

We experiment with different settings of the selection proportion ρ, which determines what percentage of the population ordered by fitness is eligible for reproduction. During selection, parents are drawn uniformly at random from T, which is the set of the $\rho\lambda$ best networks (see Section 5.2). Low selection proportions, i.e. low values of ρ, lead to high selective pressure, because parents are drawn from a smaller group of individuals with high apparent fitness. Therefore, we expect low values of ρ to work worse with small batches than high values of ρ.

Figure 5.2 shows results for increasing values of ρ at two different batch sizes and two different population sizes. Generally speaking, increasing ρ increases the validation accuracy (up to a certain degree). For a specific ρ, the validation accuracies across the four scenarios are not comparable because batch size and population size are influencing factors as well. In order to quantify the improvement through the use of low selective pressure in the different scenarios, we treat the relative difference in validation accuracies going from $\rho = 0.1$ to $\rho = 0.2$ as a proxy. Table 5.2 confirms that decreasing the selective pressure (by increasing ρ) has a positive influence on the validation accuracy, especially for small batch sizes. This is in line with our expectations, because small batch sizes lead to fitness estimates with higher variance, where keeping the selective pressure low is more important than with high batch sizes, which lead to less noise in the fitness evaluation.

5.3.4 Crossover and Mutation Operators

While the previous experiments explore the influence of limited evaluation, another significant factor for good performance are crossover and mutation operators that match the optimization problem. Neural networks in particular have problematic redundancy in their search space. Nodes in the network can be reordered without

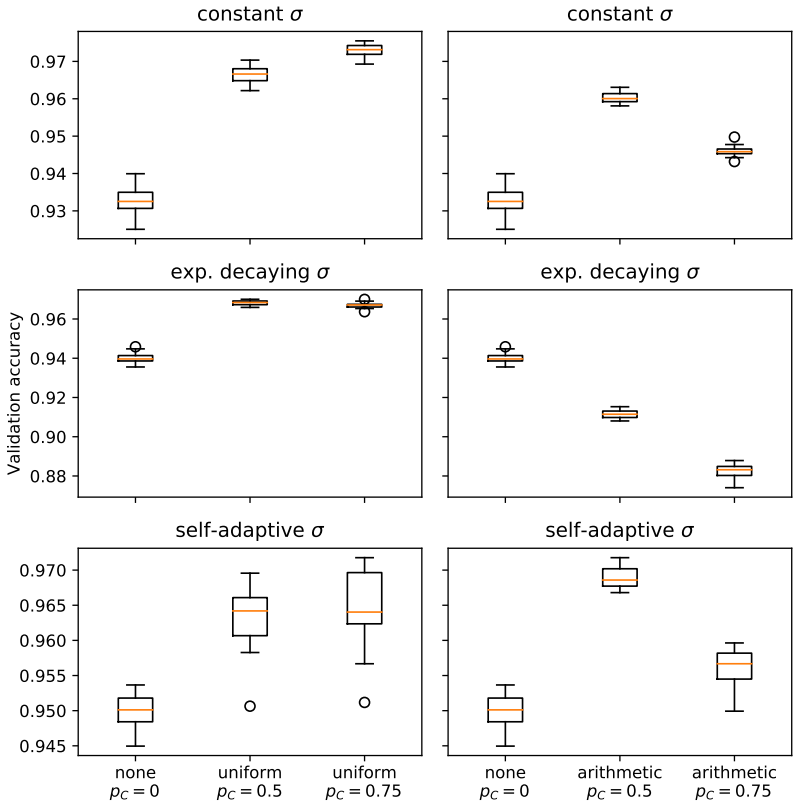

Figure 5.3: Validation accuracies of 15 EA runs with different levels of crossover p_C, crossover operators and mutation strength σ adaptation schemes. The left column shows results using uniform crossover, while arithmetic crossover is employed to get the results in the right column.

changing the network connectivity. This means that there are multiple equivalent weight vectors that represent the same function mapping.

Designing crossover and mutation operators that are specifically equipped to deal with these problems seems like a promising research direction, but for now we want to establish baselines with commonly used operators. In particular, these are uniform and arithmetic crossover as well as random normal mutation. It is not obvious if crossover is helpful for optimizing neural networks, as there is no clear compositionality in the weight space. There are many interdependencies between weights that might be destroyed, e.g. when random weights are replaced by those from another network during uniform crossover. We not only want to compare the uniform and arithmetic crossover operators among themselves, but also test if crossover leads to improvements at all. This can be achieved by varying the EA hyperparameter p_C, which controls the percentage of offspring that are created by the crossover operator.

On the other hand, random normal mutation intuitively performs the role of a local search, but its usefulness significantly depends on the choice of the mutation strength σ. Therefore, we compare the three adaptation schemes (constant, exponential decay, and self-adaptation) that provide different mutation strengths over the course of evolution.

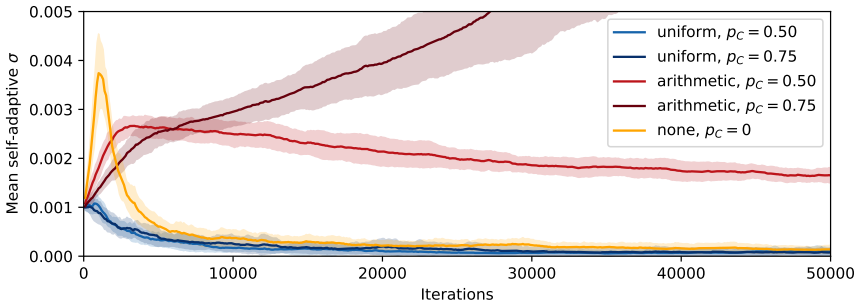

Figure 5.4: Population mean of mutation rate σ from 15 EA runs with self-adaptation turned on. The shaded areas indicate one standard deviation around the mean.

Since different crossover operators might need different mutation strengths to operate optimally, we test all combinations and show results in Figure 5.3. In experiments that use at least some amount of crossover, i.e. $p_C > 0$, validation accuracies are always significantly (Mann-Whitney U test, $p < 0.01$) higher than in experiments without crossover, except for the case of arithmetic crossover with exponential decay. The reason for this exception is likely that arithmetic crossover needs high mutation strengths to counteract the loss of diversity that is incurred by continually taking averages of solutions. In combination with exponential decay, σ decreases too fast to provide this additional diversity. This becomes evident when examining the mutation strengths chosen by self-adaptation in Figure 5.4. Compared to uniform crossover, the self-adaptation drives σ to much higher values when arithmetic crossover is used.

Overall, both crossover operators work well under different circumstances. Uniform crossover at $p_C = 0.75$ with constant σ achieves the highest median validation accuracy of 97.3 %, followed by arithmetic crossover at $p_C = 0.5$ with self-adaptive σ at 96.9 % validation accuracy. When using uniform crossover at $p_C = 0.75$, a constant mutation strength works significantly ($p < 0.01$) better than the other adaptation schemes. On the other hand, for arithmetic crossover at $p_C = 0.5$, the self-adaptive mutation strength performs significantly ($p < 0.01$) better than the other two tested adaptation schemes. The main drawback of the self-adaptive mutation strength is the the additional randomness that leads to high variance in the training results. This could probably be overcome with even bigger populations.

5.3.5 Comparison to SGD

Informed by the other experiments, we want to run the EA with advantageous hyperparameter settings and compare its test set performance to the Adam optimizer. Most importantly, we use a large population, large batch size, no fitness inheritance, and offspring are created by uniform crossover in 75 % of all cases, as summarized in Table 5.1. Population and batch size were chosen so that the population uses most of

the available GPU memory.

Median test accuracies over 15 repetitions are 97.6 % for the EA and 98.0 % for Adam. Adam still significantly ($p < 0.01$) beats EA performance, but the difference in final test accuracy is rather small. However, training with Adam progresses about 10 times faster, so it would be wrong to claim that EAs are competitive for neural network training on supervised learning problems. Yet, this work is another piece of evidence that EAs have potential for applications in this domain despite the very large search space that EAs aren't traditionally applied to.

5.4 Conclusion

Efficient batch fitness evaluation of a population of neural networks on GPUs makes it feasible to perform extensive experiments with the LEEA. While the idea of using very small batches for fitness evaluation is appealing for computational cost reasons, we find that it comes with the drawback of significantly lower accuracy than with larger batches. Furthermore, the fitness inheritance that is supposed to offset these drawbacks actually has a detrimental effect in our experiments. We propose to use low selective pressure as an alternative countermeasure.

We compare uniform and arithmetic crossover in combination with different mutation strength adaptation schemes. Surprisingly, uniform crossover works best among all tested combinations, even though it is counter-intuitive that randomly replacing parts of a network's weights with those of another network is helpful.

Finally, we train a network of 92,000 weights on MNIST using an EA and reach a median test accuracy of 97.6 %. SGD still achieves higher accuracy at 98 % and is remarkably more efficient in doing so, but this apparent drawback is mostly due to the supervised learning setting. Here, gradient-based learning works exceptionally well, whereas other areas such as reinforcement learning paint a different picture. We will take a look at an application to reinforcement learning in the next chapter. However, since in this thesis we are concerned with neural architecture search (G1) and computational efficiency (G2) on supervised image classification problems, it seems more fruitful to keep training network weights using SGD on these kinds of tasks.

Chapter 6

Application to Deep Reinforcement Learning

The last chapter demonstrates how a DNN can be trained by an EA on a supervised image classification task. While it is possible to do so, the results are not competitive with SGD training. This is not extremely surprising, because the EA completely disregards the analytical weight gradient. Therefore, the EA solves a more difficult problem compared to SGD, which uses this gradient information. However, there are other domains where EAs can outperform SGD despite this disadvantage.

Recent literature suggests that EAs achieve very competitive results on reinforcement learning tasks. In a reinforcement learning setting, an agent interacts with an environment through actions and receives rewards in response [101]. An agent can be represented by any kind of mapping between environment states and actions, i.e. it can be a lookup table but also a DNN model. The goal for the agent is to maximize the total reward over its lifetime. Because of this abstract process, many tasks can be cast as a reinforcement learning problem though it is most natural when there is an actual agent that interacts in an environment, such as in robotics.

We implement an evolutionary reinforcement learning algorithm that trains a DNN agent to play Atari 2600 games from pixel inputs. Atari 2600 games are a challenging reinforcement learning benchmark for a variety of reasons, as we will detail later. In particular, there is a lot of randomness involved that leads to high variance in scores between different evaluations of the same agent. With this observation in mind, we propose tweaks to an existing algorithm that improve its robustness in the presence of strong fitness noise.

This chapter will serve to showcase a more successful application of EAs to DNN weight training. In the reinforcement learning setting, it should be a lot more realistic to create modern neuroevolution algorithms that perform well with large DNNs on difficult problems. While we are concerned primarily with supervised image classification in this thesis, this serves to show that other domains that currently heavily rely on deep learning can also benefit from evolutionary algorithms. Reinforcement learning is just one of these domains; generative image modeling is also starting to include evolutionary approaches [138, 20, 132].

Outline. Section 6.1 introduces the Atari environment and explains the difficulty in learning to play these games. In Section 6.2 we introduce the evolutionary reinforcement learning algorithm and motivate our modifications. Section 6.3 presents experiments on a selection of six games and we summarize our work in Section 6.4.

6.1 Atari Environment

The Atari 2600 is a game console that was released in the seventies. Its dated hardware makes it easy to emulate Atari games with high speed on modern computers. This makes them good candidates as experimental testbeds for game playing, especially in the context of reinforcement learning, because the simulation of Atari environments is inexpensive compared to other environments like modern computer games. Games in general are popular to test reinforcement learning algorithms, because they often require complex skills and allow to compare the algorithms to human performance through scores. Therefore, the Arcade Learning Environment [9], which offers access to many different Atari games through a convenient Python wrapper, has become a popular benchmark for reinforcement learning.

The game state upon which the agent acts can be either the pixel image that is rendered by the game, or the game's random access memory (RAM). Since the Atari only has 128 bytes of RAM, this is a very compact representation that can easily be learned from. If the state is given as the RAM, the agent will often have direct access to meaningful properties, like the player or enemy position. In contrast, if the state is the rendered pixel image, such high-level information has to be extracted from the image through the agent itself. This complicates the learning process significantly, but it also makes the algorithms that manage to learn from such a representation transferable to other reinforcement learning problems that have visual state input. In the end, this transferability is what matters to allow real-world applications.

Despite their simple appearance, Atari games pose significant challenges to current reinforcement learning algorithms. Some games require the player to complete long sequences of actions before giving any kind of reward in the form of score points. For example, the first level of the game Montezuma's Revenge requires the player to climb down a ladder, jump over a gap, climb down another ladder, evade an enemy, and climb up another ladder to get a key. Only after the key is collected, a reward is issued. Therefore, there is no incentive for the agent to perform any of these steps until the sequence has been completed in its entirety at least once by random chance. Unlike humans, the agents are missing the prior information about ladders and keys, which gives hints on the required actions to progress in the game.

Another hurdle is non-determinism. There is randomness inherent to some games, but additional randomness is injected by the Arcade Learning Environment in the form of non-deterministic frame-skips. For some game frames, the environment simply repeats the last action instead of querying the agent for a new action to prevent

the agent from learning brittle strategies. The non-determinism usually results in a significant score variance for the same agent over repeated evaluations. In the EA context, the score is used as the fitness function, and the high variance increases the difficulty of optimization.

6.2 Evolutionary Deep Reinforcement Learning

Evolutionary deep reinforcement learning has been applied to Atari games with good results, e.g. using evolution strategies [19, 58, 107, 16] or a population based EA [125] as the DNN optimizer. EAs can easily be scaled to all available hardware resources, because they are inherently parallel. In a distributed setting, it is necessary to communicate agents and rewards over the network. With DNN agents, this would ordinarily be a bottleneck due to the large amount of weights. However, ES and EA with random normal mutation admit a strong compression of the weights by simply expressing them in terms of seeds for random number generators that create the initial weight and mutation vectors [107]. This way, the weights of a whole DNN can be expressed in just a few scalar values. The actual weights can then be reconstructed at the receiver at relatively low cost.

In comparison to Q-Learning and similar reinforcement learning algorithms, EAs are insensitive to the value distribution of rewards and long time horizons between rewards, because the fitness is aggregated over whole episodes and rank-based selection is possible. This, together with the fundamentally different optimization approach, can lead to qualitatively different solutions, e.g. with increased robustness [66].

Algorithm 3: Evolutionary algorithm for Atari DNN agent training.

$P \leftarrow$ randomly initialize λ weight vectors $\theta_1, \theta_2, \ldots, \theta_\lambda$
evaluate fitness for all solutions in P
$P \leftarrow$ best μ solutions from P
while termination condition not met **do**
  $E \leftarrow$ select κ best solutions from P
  re-evaluate and update fitness for eligible solutions in E
  $Q \leftarrow$ select λ solutions from P uniformly at random with replacement
  $Q \leftarrow \{\theta + \sigma\mathcal{N}(0,1) | \theta \in Q\}$
  evaluate fitness for all solutions in Q
  $P \leftarrow$ best μ solutions from $E \cup Q$
end

We implement an EA for deep reinforcement learning heavily inspired by Such et al. [125] with a tweak that increases its robustness against fitness noise. It is a population-based (μ, λ) EA with elitism and is listed in Algorithm 3. Agents are trained for every game separately, i.e. they can only play a single game. The training

begins by initializing λ random DNN agents, evaluating them, and choosing the best μ as the initial parents P. Then, the evolutionary loop starts and, in our case, continues until the number of total processed game frames by all fitness evaluations so far exceeds a set limit. Every iteration, we select the κ best agents by fitness from P as elites, which are not mutated in any way. Furthermore, we select λ random solutions from P, perform random normal mutation with mutation strength σ, and evaluate these offspring agents.

During the evaluation each agent is given a $4 \times 84 \times 84$ tensor as its input state that consists of the last four grayscale game frames stacked on top of each other. This allows the agent to perceive motion without recurrent connections in the DNN. The agent plays the game until it dies or exhausts a predetermined budget of 3,000 frames. This prevents agents that perform nonsensical actions such as always standing still from never completing their evaluation. Our frame budget is set rather low because of computational resource constraints, and this would likely prevent state-of-the-art results for some games. Nevertheless, good results can be achieved even in this scenario.

Every evaluation is performed using a different random seed, and a random number of no-ops (up to 30) are performed at the beginning of every episode, which is a common configuration to make the agent experience different states at the beginning of the game. This in turn increases the difficulty, because it requires more adaptive behavior from the agent. The score achieved by the agent at the end of the episode is returned as its fitness and often varies strongly between evaluations of the same agent due to the non-determinism.

Such et al. [125] select μ solutions among elites and offspring as parents for the next generation. They use only a single elite ($\kappa = 1$) and select a total of $\mu = 10$ parents. In our experiments, this often leads to the parent population being filled with agents that have a spurious high score run but perform badly in repeated evaluations. Agents that perform well consistently but did not score exceptionally high are very likely to be replaced by such bad agents with apparent high fitness.

Therefore, we tweak the algorithm in several ways. First of all, we increase the number of elites to $\kappa = 10$ and size of the parent population to $\mu = 40$. When selecting a larger number of parents from the population, it becomes increasingly unlikely that all parents will be badly-performing agents with overestimated fitness, i.e. there is a higher chance for offspring to derive from actually good agents. Increasing the number of elites also decreases the chance that all of them will have an overestimated fitness value, although the chance is still higher because elites are specifically the agents with highest fitness. Therefore, we introduce a re-evaluation procedure (see Algorithm 3, line 6). Re-evaluation is a common countermeasure to fitness noise that performs multiple evaluations of the same solution and averages the result. This increases the computational cost, so we do not re-evaluate the whole population but only the elites, which are most prone to contain agents with overestimated fitness. Every generation,

Table 6.1: Hyperparameters for the baseline and our proposed evolutionary reinforcement learning algorithm.

Hyperparameter	Baseline	Proposal
Elites κ	1	10
Parents μ	10	40
Offspring λ	5,000	2,000
Mutation strength σ	0.005	0.005
Max. evaluations	1	20

an additional re-evaluation is performed (until a solution reaches a maximum number of evaluations), so that the elite agents' fitness values approach their true fitness value over time.

6.3 Experiments

We train DNN agents on six different Atari games using the EA defined in Algorithm 3. Both the baseline from Such et al. [125] and our algorithmic tweak can be expressed by choosing different hyperparameters, as specified in Table 6.1. In addition to the increased number of elites and parents, we allow up to 20 evaluations per agent. In consequence, elites are re-evaluated up to 19 times, and their fitness gets more accurate with each re-evaluation. Setting the maximum number of evaluations to 1 recovers the original algorithm from Such et al [125].

The DNN modeling our agents consists of three convolutional layers and two dense layers, each followed by a ReLU activation except for the final layer. The convolutional layers in sequence are configured as follows: 32 channels using a 8×8 kernel with stride 4; 64 channels using a 4×4 kernel with stride 2; 64 channels using a 3×3 kernel with stride 1. The first dense layers has 512 outputs, while the final dense layer has a game-dependent number of outputs that corresponds to the number of buttons that the game uses. For the game Atlantis there are 4 outputs, while there are 18 outputs for all other games. This architecture is identical to the one Such et al. [125] use. The action corresponding to the output with the highest score is chosen by the agent at each timestep.

The evolution progresses until a total of one million game frames have been processed. Unfortunately, it is difficult to compare results between different reinforcement learning algorithms directly, as the different algorithms use frames in a different way. For example, gradient-based methods put frames in a replay memory, and frames are re-used multiple times to train the DNN. In addition to the cost of using an agent to play the game in order to receive new state transitions, the gradient-based methods incur training costs. Therefore, we can only directly compare between the baseline variant and our proposed variant of Algorithm 3. We will still list results of other

Table 6.2: Results on six Atari games from literature and our own experiments. The asterisk marks experiments that use ensembling.

Method	Atlantis	Frostbite	Gravitar	Kangaroo	Seaquest	Venture
DQN [82]	279,987	797	473	7,259	5,861	163
Duel [139]	382,572	4,673	588	**14,854**	**50,254**	497
ES [107]	**1,267,410**	370	**805**	11,200	1,390	760
Baseline	57,294	5,547	200	8,213	1,316	626
Baseline*	45,168	5,140	124	9,850	628	0
Proposal	96,057	**6,440**	656	10,032	1,369	**1,230**
Proposal*	100,987	2,455	480	10,198	921	580

approaches from literature to provide context.

For every experiment, we retrieve the single best agent from the union of all generations according to their fitness values. For a fair comparison between the baseline and our proposal, the best agents are evaluated by playing their game for 100 trials. Each trial has a fixed but unique seed, so that the environment is the same for both agents. Figure 6.1 shows how the evolution progresses for both baseline and proposal. We see that the proposed algorithm performs similarly to the baseline in the game Seaquest and outperforms the baseline in the five other games.

Final scores are listed again in Table 6.2, together with results from the differentiable deep reinforcement learning algorithms DQN [82], Duel [139], and another method based on evolution strategies (ES) [107]. We can see that the different algorithms excel at different games. Our proposed EA scores highest in two games, the ES scores highest in another two games, and Duel scores highest in the remaining two games. Overall, evolutionary methods (EA and ES) are very competitive with gradient-based approaches.

However, the state-of-the-art has since advanced significantly by exploiting scaling effects. For example, the recently published Ape-X [41] employs a training architecture distributed among a large number of workers and uses prioritized experience replay [111] to select the most promising transitions for training the agent. Despite only combining already existing deep reinforcement learning techniques, they significantly advance the state-of-the-art, because their method is able to generate and exploit several orders of magnitude more state transitions as training data than before. It stands to reason that evolutionary approaches would similarly benefit from more samples.

Finally, we test if it is possible to ensemble the agents that get created by evolution for even better results. This would be a great side-effect of EAs, since they naturally train a population of well-performing agents. To create an ensemble, we take the best three unique agents from the union of all generations according to their fitness values. The outputs of all models are averaged before the agent chooses its action. Results are

listed in Table 6.2 as the experiments marked with an asterisk. There are some slight improvements in some games but also large losses in other games. Overall, this kind of ensembling does not seem promising out of the box. A possible explanation might be that the different agents follow different strategies that contradict each other but we did not investigate this further.

6.4 Conclusion

We implement an EA to train DNNs in a reinforcement learning setting using Atari games as a benchmark problem. Reinforcement learning poses a very different challenge from supervised learning and is usually strongly affected by noise from various sources. EAs can be designed in ways that makes them very resilient to noise and we have demonstrated this by improving upon an existing evolutionary deep reinforcement learning method.

EA and ES perform competitively to gradient-based methods on Atari games although each method seems to excel at different tasks. Due to the fundamentally different algorithmic approach between gradient-based Q-learning and evolution, we can expect very different solutions with different properties and this allows for promising future work.

This chapter showcases that evolutionary techniques can be used to train large DNNs for complex problems just as well as SGD even if they cannot compete with SGD on supervised image classification tasks. For the remainder of the thesis will return to supervised learning, but we want to point out that reinforcement learning seems like a very promising domain to test modern neuroevolutionary approaches that use large DNNs.

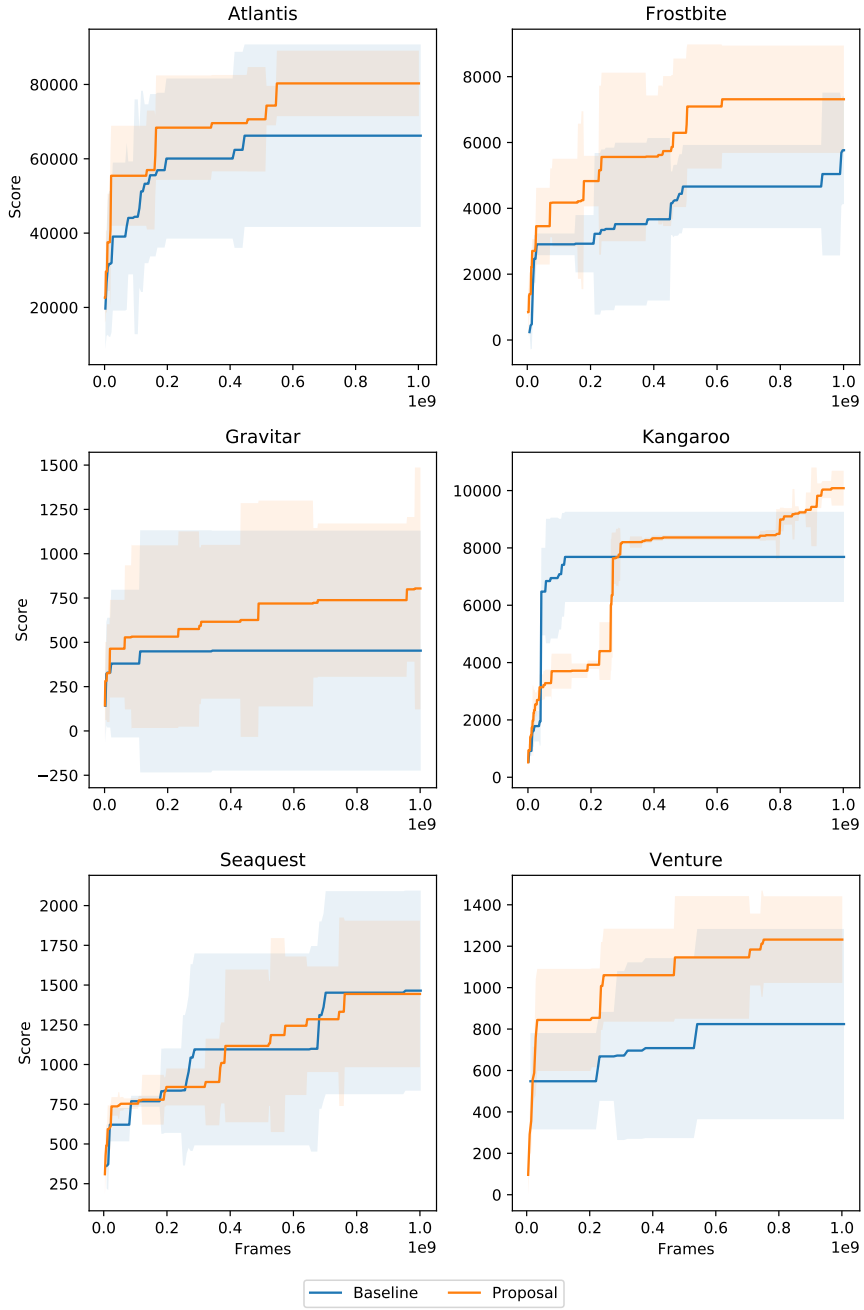

Figure 6.1: Progress of evolution in terms of score over total processed number of frames. Shown is the average and standard deviation over 50 trials of the best agent in each generation.

Part III

Evolutionary Neural Architecture Search

After exploring DNN weight training with EAs, we now turn our attention to the DNN architecture. So far, the architecture has been manually set beforehand, but it is difficult and inefficient to manually search for good architectures in the enormous space of possibilities. Since we saw that EAs are not competitive with SGD for the training of DNNs on supervised learning problems, we will now explore neural architecture search methods that train weights with SGD while optimizing the architecture with evolutionary algorithms. Chapter 7 will introduce evolutionary NAS approaches and explore a technique to reduce the computational requirements of NAS. Then, we turn to one-shot NAS for even larger efficiency gains. In Chapter 8, we present a NAS algorithm that closely integrates the SGD weight training process with an evolutionary optimization of the architecture by a natural evolution strategy. The algorithm is used to perform deep multi-task learning with the goal to find architectures that allow the application of DNNs to multiple small but related tasks.

Chapter 7

Lamarckian Evolution of Convolutional Neural Networks

Convolutional neural networks, i.e. DNNs with convolutional layers as their main component, perform extremely well as image classifiers. As we have seen in Section 2.4, the architecture of the network has significant impact on its performance. Neural architecture search methods allow to automatically find well-performing architectures, but the process is computationally very expensive, sometimes requiring thousands of GPU-hours.

One of our goals in this thesis is to reduce the computational requirements for neural architecture search. In this chapter, we show that an evolutionary algorithm saves training time during the network architecture optimization if learned network weights are inherited over generations by Lamarckian evolution. By using a parent's trained weights as the starting point for the optimization of an offspring solution, less training time is required for the evaluation of the offspring. Experiments on typical image datasets show similar or significantly better test accuracies and improved convergence speeds compared to two different baselines without weight inheritance. On CIFAR-10 and CIFAR-100, weight inheritance reduces the necessary number of training epochs to match or surpass baseline accuracies by 75 %.

The results presented in this chapter are based on the following publication:

- Jonas Prellberg and Oliver Kramer. Lamarckian evolution of convolutional neural networks. In Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and Darrell Whitley, editors, *Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part II*, volume 11102 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 2018

Outline. Section 7.1 presents related work about approaches that optimize DNN architectures with EAs. In Section 7.2, we detail EA and non-EA approaches that have used weight inheritance for various reasons. Then, Section 7.3 describes the EA that is used in this chapter and explains how the mutation with weight inheritance works. In Section 7.4, experimental results are presented and discussed. We end with a conclusion in Section 7.5.

7.1 Evolutionary Neural Architecture Search

Over the last years, DNNs and especially convolutional neural networks (CNN) have become state-of-the-art in numerous application domains. Their performance is greatly influenced by the network architecture, which makes choosing the right architecture an important aspect of applying neural networks to new problems. However, comparing different architectures is computationally expensive due to the lengthy training process that has to be repeated each time a new architecture is tested. This downside applies to optimization by hand as well as to automated approaches, such as grid search, random search, or evolutionary optimization.

In contrast to neuroevolutionary approaches that search architectures in the space of individual nodes and connections, in the recent years EAs have mostly been applied to optimize network architectures on a coarser level. They work on the level of building blocks or hyperparameters that describe regular architectures. The actual weight training is performed with SGD, since it offers increased efficiency for training the large and deep networks prevalent today. The usual procedure is as follows: A genotype encodes the network architecture, e.g. as a graph or a list of building blocks. In a genotype-phenotype process, a network is built from this description and initialized with random weights. Then, several epochs of SGD on a training set adjust the network weights. Finally, the network is tested on a validation set and a metric, such as accuracy, is reported as the solution's fitness.

There are numerous publications that follow this general NAS concept. Kramer [60] uses a $(1 + 1)$ EA with a special niching strategy to optimize hyperparameters that define a convolutional highway network, a specific human-designed architecture that can be configured in different ways. Suganuma et al. [126] use a modified $(1 + \lambda)$ EA to optimize the structure of a CNN using a Cartesian genetic programming encoding scheme. Both approaches use small populations and only employ mutations while still achieving good results. Desell [23] modifies the NEAT algorithm to optimize the structure of a CNN by evolving a graph of convolutional layers which is then trained with SGD. Similar to Desell, CoDeepNEAT [80] also extends the NEAT algorithm to use DNN layers as nodes in the evolved graph. Additionally, a co-evolution of modules and blueprint networks is performed. The blueprint networks reference the evolved modules, which results in architectures with repeated motifs similar to how hand-designed architectures are mostly built from repeated blocks of the same type. Another recent approach, DENSER [5], uses a genetic algorithm to optimize a sequence of neural network layers whose hyperparameters are evolved through dynamic structured grammatical evolution.

All of these approaches have one thing in common: Over the course of the evolution, thousands of networks are trained using SGD, which is extremely resource intensive. As we will see next, weight inheritance can offset some of the training cost.

7.2 Weight Inheritance

Lamarckian evolution describes the idea that traits acquired over the lifetime of an individual are inherited to its offspring [110]. While rejected in biology, this approach can be beneficial for artificial evolution when there is a bi-directional mapping between genotype and phenotype. This allows to encode learned behavior back into the genotype and then apply an EA as usual. For example, Parker and Bryant [88] and Ku et al. [62] apply Lamarckian evolution to neural networks by directly encoding the network weights in the genotype. This creates a simple one-to-one mapping between genotype and phenotype. Other publications also use weight inheritance with more or less success and different goals.

Desell [23] conducted an experiment on weight inheritance in the context of their NEAT CNN training algorithm, but it was not found to decrease the time necessary to train a single network to completion.

Fernando et al. [26] use a microbial genetic algorithm to optimize the structure of a DPPN. A DPPN is a differentiable version of the compositional pattern producing network that is also used in HyperNEAT and produces weights for a second target network. Fernando et al. found their Lamarckian EA with weight inheritance to improve the mean squared error in an image reconstruction experiment over their EA without weight inheritance.

Jaderberg et al. [49] use an EA to optimize hyperparameters of static networks. It only performs mutations but inherits weights to mutated offspring. The networks are therefore trained to completion over multiple steps with potentially different hyperparameters. If one of the optimized hyperparameters is the learning rate, this effectively trains the network using a dynamic, evolved learning rate schedule. Different from their work, our goal is to increase the data efficiency of a NAS algorithm.

Real et al. [98] use an EA to optimize the structure of a CNN through mutations with weight inheritance. During fitness evaluations, SGD training is performed for 28 epochs on CIFAR-10 or CIFAR-100. Using very large population sizes, they reach results that are competitive with hand-designed networks. Weight inheritance is not a focus of their work, but they show that it is necessary to evolve networks with good final accuracy, since 28 epochs are not enough to fully train a network on the used datasets.

We will look at weight inheritance in more detail in this chapter and show that a weight inheritance scheme can drastically increase the data efficiency of an EA that optimizes neural network architectures. Usually there is a trade-off between training time per fitness evaluation and final accuracy. The longer each network is trained during its fitness evaluation, the more accurate the fitness estimate gets. However, the total runtime of the EA increases significantly as well. Weight inheritance allows to resolve this trade-off by increasing the convergence speed of network training through a better-than-random weight initialization.

7.3 Lamarckian Neural Architecture Search

To assess the influence of weight inheritance for neural network architecture optimization, design decisions regarding the optimizable architecture search space and type of EA must be made. The goal is not to achieve state-of-the-art performance or find novel architectures but instead to show the advantages of weight inheritance. Therefore, we choose to optimize a fairly restricted architecture space of linearly stacked building blocks which, however, is still applicable to many problems.

In contrast, let us consider NAS approaches like [154, 155, 69] that allow a wider variety of network architecture graphs because nodes can connect to several input and output nodes. This comes with drawbacks such as solutions that do not represent valid network architectures due to incompatibilities in data shapes or nodes without any inputs or outputs that have to be heuristically connected. Most importantly though, this creates a significantly larger search space that covers a larger number of possible architectures. While this is advantageous in general because some of these architectures may be better than what can be found in a more constrained search space, it becomes a disadvantage if only a tiny fraction of it can be explored. Since we are concerned with NAS in a resource-constrained setting, the total number of explored architectures will be comparatively small. In such a scenario, it is more beneficial to explore a smaller, coarse-grained search space well in order to cover a diverse set of architectures. In contrast, exploring solutions in a small part of a fine-grained solution space would waste computation on many similar architectures. Furthermore, this choice allows the EA to converge fast enough within our hardware resource constraints to make multiple repetitions of the same experiment feasible for statistical purposes.

The architecture search space is defined by the template presented in Figure 7.1. It consists of stacked building blocks that contain a convolutional layer followed by batch normalization and a ReLU activation. The total number of building blocks and the individual number of channels, kernel size, and stride of the convolutional layer in each building block are subject to optimization. They make up the genotype for this evolutionary optimization. When creating the phenotype, i.e. the actual network, we always append global average-pooling and a dense layer after the last building block, since experiments are performed specifically on image classification datasets.

The optimization is performed by a $(1 + 1)$ EA. In contrast to evolutionary algorithms with larger populations, the necessary computational resources are modest, but the method is also more prone to getting stuck in local optima in multi-modal problems. To alleviate this, a form of niching is introduced. The evolutionary algorithm and its mutation operator with weight inheritance are presented in more detail in the following sections.

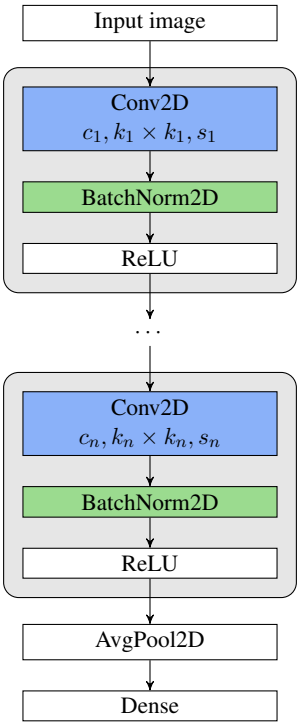

Figure 7.1: Graph template defining the network architecture search space, which is a sequence of building blocks (shown as gray boxes). Each building block $i \in [\![n]\!]$ has its individual hyperparameters channel count c_i, kernel size k_i, and stride s_i that have to be optimized by the EA.

7.3.1 Evolutionary Algorithm

In contrast to all previous descriptions of the DNN training process in Part I and II, we now not only optimize weights but also the DNN architecture. Therefore, we now minimize an extended loss function $\mathcal{L}(\theta, a) : \Theta \times \mathcal{A} \to \mathbb{R}$ that depends on both weights and architecture, where Θ is the space of weights and $\mathcal{A}$ is the space of DNN architectures. Our method requires the dataset to be split into a training set D_{train} and validation set D_{val}. The loss function $\mathcal{L}(\theta, a)$ will only use data from D_{train} and is used for weight training with SGD while keeping the architecture fixed. We further define a fitness function $\mathcal{V}(\theta, a)$, e.g. accuracy for classification problems, that only uses data from D_{val} and is used to guide the evolutionary architecture optimization.

Algorithm 4 presents the $(1+1)$ EA with niching as pseudo-code. An initial solution (θ, a) with an architecture a consisting of a single convolutional layer with a random number of channels, random kernel size, and a stride of one is created and initialized with random weights θ. This solution is optimized by the EA in a loop as follows.

First, a random mutation from the set of possible mutations is applied to the parent (θ, a) to create a child (θ', a') with a different network architecture but inherited weights. Note that θ' will not generally be equal to θ, even though that is the intent behind weight inheritance. Changes in the architecture will lead to differently shaped

Algorithm 4: $(1 + 1)$ evolutionary algorithm for NAS with niching.

Let $\mathcal{L}(\theta, a)$ be the loss on the training set under weights θ and architecture a
Let $\mathcal{V}(\theta, a)$ be the fitness on the validation set under weights θ and architecture a
def `EvoNAS` $(\theta, a, \texttt{steps}, \texttt{allowNiching})$ **as**
    for i in $1 \dots \texttt{steps}$ **do**
        $\theta', a' \leftarrow$ apply mutation with weight inheritance to (θ, a)
        $\theta^* \leftarrow$ minimize $\mathcal{L}(\theta', a')$ for e epochs by SGD with a' fixed
        if `allowNiching` and $\mathcal{V}(\theta^*, a') \leq \mathcal{V}(\theta, a)$ and $\texttt{rnd}() < \omega$ **then**
            $\theta^*, a' \leftarrow$ `EvoNAS` $(\theta^*, a', k, \texttt{False})$
        end
        if $\mathcal{V}(\theta^*, a') > \mathcal{V}(\theta, a)$ **then**
            $\theta, a \leftarrow \theta^*, a'$
        end
    end
    return (θ, a)
end
$\theta, a \leftarrow$ initial arch. with a single random building block and random weights
$\theta, a \leftarrow$ `EvoNAS` $(\theta, a, \texttt{maxSteps}, \texttt{True})$

weight tensors that make inheritance impossible in some cases. It will however be as similar as possible and keep weights of all building blocks that are unaffected by the mutation. These mutations with weight inheritance are described in the following sections.

Next, the weights θ' of this child solution are optimized by SGD for e epochs by minimizing $\mathcal{L}(\theta', a')$ while keeping a' fixed. The resulting trained weights θ^* are used to evaluate the child solution's fitness $\mathcal{V}(\theta^*, a')$. If it is greater than the parent's fitness $\mathcal{V}(\theta, a)$, the child replaces the parent.

Because this algorithm is greedy, it can get stuck in local minima. Therefore, a niching approach adapted from Kramer [60] is implemented. There is a random chance ω to follow solutions that are initially worse. In such a case, a child, which has a lower fitness than its parent, is used as the parent network for a recursive call of the same algorithm. During niching, the mutate-evaluate-select-loop is repeated k times. When the last loop iteration ends, the best network found during niching is returned. If this network has a greater fitness than the original parent, it is selected. Otherwise, optimization proceeds with the original parent.

7.3.2 Mutation Operators

As mentioned before, the number of building blocks, as well as the number of channels, kernel size, and stride of each convolutional layer are subject to optimization. For simplicity, all these hyperparameters are chosen from predefined sets:

- Number of building blocks $n \in \mathbb{N}$
- Number of channels $c_i \in \mathcal{C} = \{16, 32, 64, 96, 128, 192, 256\}$ for all $i \in [\![n]\!]$
- Kernel sizes $k_i \in \mathcal{K} = \{1, 3, 5\}$ for all $i \in [\![n]\!]$
- Strides $s_i \in \mathcal{S} = \{1, 2\}$ for all $i \in [\![n]\!]$

Mutations are picked randomly from the list below. Each choice has a relative frequency (indicated by the multiplier in front of the list item) that determines how much more likely it is to be chosen than the mutation with a relative frequency of one. The frequencies have been chosen such that the more granular mutations, which are likely to have a smaller impact on the result, are applied less often in order to effectively use the available computation time.

- $3 \times$ *add block*: Adds a building block at a random position. The contained convolutional layer is initialized with a random number of channels, random kernel size, and a stride of one. This mutation indirectly governs the number of building blocks n.
- $3 \times$ *remove block*: Removes a random building block. This mutation indirectly governs the number of building blocks n.
- $2 \times$ *add channels*: Picks a random convolution and sets its number of channels to the next greater value in $\mathcal{C}$.
- $2 \times$ *remove channels*: Picks a random convolution and sets its number of channels to the next lower value in $\mathcal{C}$.
- $2 \times$ *change kernel size*: Picks a random convolution and randomly draws its kernel size from $\mathcal{K}$.
- $1 \times$ *change stride*: Picks a random convolution and randomly draws its stride from $\mathcal{S}$.

All random choices within each mutation, such as picking a random block to apply the mutation to or picking a new kernel size, are chosen uniformly at random from the appropriate set.

The mutation operator is forced to modify the network. A history of all previously evaluated network architectures is maintained, and mutations are repeatedly applied to the parent network until an architecture is created that has not been evaluated before. Furthermore, only network architectures with at most three convolutions of stride two are allowed because the image inputs of CIFAR are only 32×32 and each stride-two convolution halves the side lengths.

7.3.3 Weight Inheritance

Each solution consists of a network architecture made from n building blocks and the associated weight vector. The latter is the concatenation of all weights in all building blocks that are part of the corresponding network architecture:

$$\theta = \theta_1 \parallel \ldots \parallel \theta_n. \tag{7.1}$$

Here, $\parallel$ denotes the concatenation of flattened tensors. The weights θ_i for building block i are comprised of the convolutional layer kernel $\mathbf{k}_i \in \mathbb{R}^{c_i \times c_{i-1} \times k_i \times k_i}$ and bias $\mathbf{b}_i \in \mathbb{R}^{c_i}$, and batch normalization weights $\gamma_i, \beta_i \in \mathbb{R}^{c_i}$ and running statistics $\sigma_i^2, \mu_i \in \mathbb{R}^{c_i}$. The concatenation of weights for building block i is therefore:

$$\theta_i = \mathbf{k}_i \parallel \mathbf{b}_i \parallel \gamma_i \parallel \beta_i \parallel \sigma_i^2 \parallel \mu_i = \mathbf{k}_i \parallel \mathbf{h}_i. \tag{7.2}$$

We aggregate all one-dimensional tensors with c_i components into the tensor $\mathbf{h}_i$ for notational convenience, because they can be treated identically by all mutation operators.

When creating the initial parent network, its weights are randomly initialized in an appropriate fashion, e.g. Glorot-uniform [30] initialization. However, once a network has been evaluated, its weights contain useful, learned values. These learned weights can be exploited as a good starting point instead of random weights when an offspring solution needs to be trained for its own evaluation. By performing mutation with weight inheritance the offspring network converges faster, and the whole NAS process is accelerated.

Given a parent solution with weight vector θ, we apply one of the previously described mutation operators to the architecture. At the same time, we create a weight vector θ' that matches the mutated architecture and keeps as many components of θ as possible. Whenever a mutation changes a block i in such a way that one of the contained weight tensors $\mathbf{k}_i$ or $\mathbf{h}_i$ changes in shape, it needs to be randomly reinitialized. Other unaffected weights can be copied from the parent. However, some mutations have an effect on the block's number of output channels c_i, which determines the number of input channels for the next block $i + 1$. This requires changing the convolutional kernel $\mathbf{k}_{i+1}$ to accept c_i input channels. In such a case, $\mathbf{k}_{i+1}$ also needs to be reinitialized. All mutations are shown graphically in Figure 7.2 and are specified formally in the remainder of this section.

When the mutation *change stride* is applied to block i, all weights can be reused because the hyperparameter s_i does not modify the shape of any component in θ and it does not affect the number of output channels c_i either. Even though the spatial resolution of the output halves, this does not cause any problems. All convolutional layers can deal with arbitrary spatial input sizes and global average-pooling is performed before the dense layer, i.e. only the number of channels c_n of the last convolutional

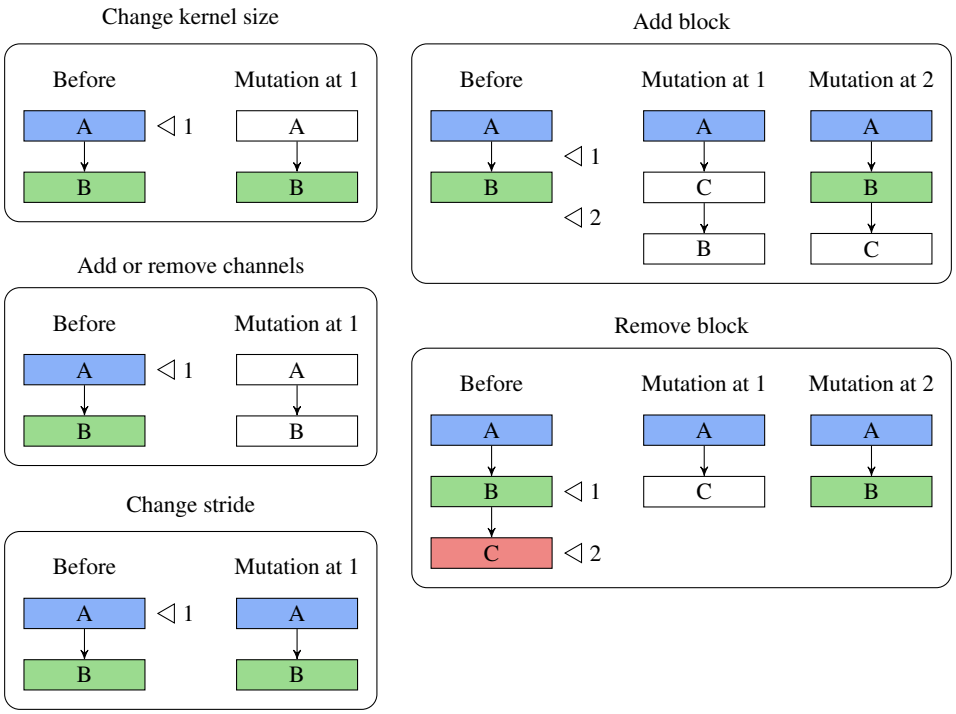

Figure 7.2: Mutation operators with weight inheritance applied to small architectures at different positions. The colors signify different weights and white blocks are reinitialized.

layer determines how many components the input tensor to the dense layer contains. In summary, the weight inheritance for *change stride* is simply implemented by $\theta' = \theta$.

The mutation *change kernel size* applied to block i affects only the weights $\mathbf{k}_i$ of the mutated block itself, which are randomly reinitialized. The convolutional layer's output shape stays identical because zero-padding is performed to keep the spatial dimensions of constant size, and the number of output channels does not change either. The new weight vector under this mutation is

$$\theta' = \theta_1 \parallel \ldots \parallel \left(\mathbf{k}_i^{\circlearrowright} \parallel \mathbf{h}_i\right) \parallel \ldots \parallel \theta_n, \tag{7.3}$$

where we use the superscript $^{\circlearrowright}$ to denote weights that are reinitialized by the same random initialization method that was used to set the initial network weights.

Applying the mutations *add channels* or *remove channels* to block i requires both $\mathbf{k}_i$ and $\mathbf{h}_i$ to be reinitialized because both tensors' shapes are affected by c_i. Furthermore, the number of output channels of block i changes so that the convolutional kernel $\mathbf{k}_{i+1}$ in the following block $i+1$ also needs to be adapted to its new number of input channels and consequently reinitialized. (Unless i is the last block of course, i.e. block $i+1$ does not exist.) The new weight vector under either of these mutations is

$$\theta' = \theta_1 \parallel \ldots \parallel \theta_i^{\circlearrowright} \parallel \left(\mathbf{k}_{i+1}^{\circlearrowright} \parallel \mathbf{h}_{i+1}\right) \parallel \ldots \parallel \theta_n. \tag{7.4}$$

Similarly, the mutations *add block* and *remove block* lead to changes in the input shape of the block that follows the newly inserted block or that followed the block that was just removed. Adding a new block with randomly initialized hyperparameters after block i results in a new weight vector

$$\theta' = \theta_1 \parallel \ldots \parallel \theta_i \parallel \left(\mathbf{k}^{\circlearrowleft} \parallel \mathbf{h}^{\circlearrowleft}\right) \parallel \left(\mathbf{k}_{i+1}^{\circlearrowleft} \parallel \mathbf{h}_{i+1}\right) \parallel \ldots \parallel \theta_n, \tag{7.5}$$

where $\mathbf{k}$ and $\mathbf{h}$ are the randomly initialized weights of the newly inserted block. Their shape depends on the randomly set hyperparameters. On the other hand, removing the block i results in the weight vector

$$\theta' = \theta_1 \parallel \ldots \parallel \theta_{i-1} \parallel \left(\mathbf{k}_{i+1}^{\circlearrowleft} \parallel \mathbf{h}_{i+1}\right) \parallel \ldots \parallel \theta_n, \tag{7.6}$$

where the successive convolutional kernel $\mathbf{k}_{i+1}$ needs to adapt to its new number of input channels c_{i-1}.

In this description we have left out the weights of the final dense layer, which are also inherited, for brevity. They only need to be reinitialized if c_n changes and are copied in all other cases. Note that c_n may change indirectly if a new building block is appended to the end, or the last block is removed.

7.4 Experiments

Training neural networks for image classification typically takes lots of resources and improving data efficiency in this kind of setting would be of great value. Therefore, we choose to experiment on the standard image benchmarks CIFAR-10 and CIFAR-100. Both contain 60,000 RGB images of 32×32 pixels and $C = 10$ or $C = 100$ classes respectively. The datasets are split into a training set D_{train} of 45,000 examples, validation set D_{val} of 5,000 examples, and test set D_{test} of 10,000 examples for both CIFAR-10 and CIFAR-100.

For every solution (θ, a) that is created by the EA, we can construct a DNN model $f(\theta, a, \mathbf{x}) : \Theta \times \mathcal{A} \times \mathcal{X} \to \mathcal{Y}$ with weight space Θ, architecture space $\mathcal{A}$, input space $\mathcal{X} = \mathbb{R}^{3\times32\times32}$ and output space $\mathcal{Y} = \mathbb{R}^C$. During every fitness evaluation, its weights θ are trained by the Adam optimizer for e epochs with data from D_{train}. The batch size is 512 and the learning rate is 10^{-3} for this training process. After training, the validation set accuracy over inputs $\mathbf{x}^{(i)} \in \mathcal{X}$ and labels $y^{(i)} \in [\![C]\!]$ from D_{val} is calculated as

$$\mathcal{V}(\theta, a) = \frac{1}{|D_{\text{val}}|} \sum_{i=1}^{|D_{\text{val}}|} \delta_{\hat{y}^{(i)}, y^{(i)}}, \tag{7.7}$$

where $\delta_{m,n}$ is the Kronecker delta function that is 1 if $m = n$ and 0 otherwise and $\hat{y}^{(i)} = \arg\max_j \hat{\mathbf{y}}_j^{(i)}$ is the index of the highest scoring class in the network

output $\hat{\mathbf{y}}^{(i)} = f\left(\theta, a, \mathbf{x}^{(i)}\right)$. This validation set accuracy $\mathcal{V}(\theta, a)$ is used as the solution's fitness.

The test set is only used after all experiments have finished. During evolution the best solution is saved in regular intervals. After the EA finishes, these checkpoints are trained to completion for another 30 epochs on the training set using a learning rate schedule (10^{-3} until epoch 10, 10^{-4} until epoch 20 and 10^{-5} until epoch 30). After this additional training step, they are evaluated on the test set. We use this to compare test accuracies between experiments at one point during the evolution and after the evolution is finished.

The experiments are repeated 20 times with different random seeds to account for variance introduced by the randomness that is inherent to the EA and also the network training. Using the results from these repetitions, we perform significance testing using the one-sided Mann-Whitney U test.

7.4.1 Study Setup

The mutation operator that employs weight inheritance is compared to a mutation operator that randomly reinitializes all network weights after each mutation. Otherwise, the same EA with the same hyperparameters is used on both datasets. The niching rate and depth are set to $\omega = 0.1$ and $k = 5$ respectively.

During each fitness evaluation, a network is trained for e epochs and subsequently its performance is assessed on the validation set. Choosing e is a trade-off between evaluation speed and the accuracy of the fitness assessment. If e is very low, evaluation is fast, but networks are not trained to completion. Therefore, the reported fitness will usually be lower than what the network could actually achieve given enough training time. Consequently, large but accurate networks have difficulty competing with smaller networks which train faster but might reach a lower final accuracy. If e is very high, these problems vanish, but the evaluation takes a long time. Since many evaluations are necessary for large search spaces, this is impractical. Weight inheritance is supposed to offset some of the problems that come with the choice of a small training epoch budget e during fitness evaluation.

The EA gets a budget of n total training epochs as its termination condition. Every time the EA performs training steps, whether niching or not, this counts towards the budget. This allows for a comparison of accuracy in terms of training examples that each experiment has processed. Given the total epoch budget n, the choice of e influences how many generations are possible before the EA finishes. Consequently, this determines how many different architectures are tested during evolution.

We propose an EA with weight inheritance and $e = 4$ training epochs per fitness evaluation. Note that four epochs is not sufficient to train networks that work well on CIFAR to completion. The comparison baselines are EAs that do not use weight inheritance with two different epoch settings.

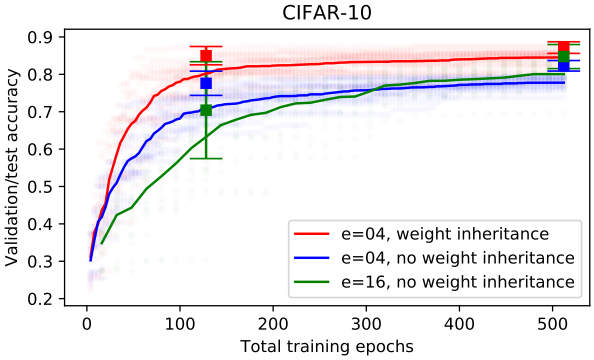

Figure 7.3: Training progression of the EA with weight inheritance compared to both baselines on CIFAR-10.

Table 7.1: Test accuracies at two checkpoints on CIFAR-10.

		128 total epochs			512 total epochs		
Inheritance	e	Min.	Mean	Max.	Min.	Mean	Max.
Yes	4	**79.1**	**85.0 ± 2.4**	**89.0**	**83.3**	**87.2 ± 1.5**	**89.3**
No	4	68.3	77.6 ± 3.2	81.8	78.9	82.3 ± 1.4	84.2
No	16	32.5	70.4 ± 12.6	85.5	76.6	84.8 ± 3.1	88.9

The first baseline, which will be called baseline I, also uses $e = 4$ training epochs per evaluation in order to allow for a direct comparison. This allows us to show that the algorithm with weight inheritance is more data efficient and has better final accuracy all else being equal.

The second baseline, which will be referred to as baseline II, uses $e = 16$ training epochs per evaluation. This significantly longer training time is more in line with the traditional approach of optimizing neural network architectures. It allows us to show that our observations still hold here, and we do not simply trade a lower final accuracy for data efficiency.

7.4.2 Results

Figures 7.3 and 7.4 compare three experimental settings on CIFAR-10 and CIFAR-100 with a total epoch budget of 512. The EA with weight inheritance outperforms the comparison baselines that do not use weight inheritance on both datasets. The accuracy plateau is reached more quickly and higher test accuracy is achieved.

In these figures, each dot in the background represents the best validation accuracy achieved so far during a repetition of the experiment. Each line shows the mean validation accuracy over all repetitions at each epoch. The boxplots show the average test accuracy after training the network checkpoints to convergence and their whiskers represent one standard deviation of these final test accuracies.

Tables 7.1 and 7.2 list minimum, mean, and maximum test accuracies of the CIFAR-10 and CIFAR-100 experiments for specific checkpoint epochs. When weight

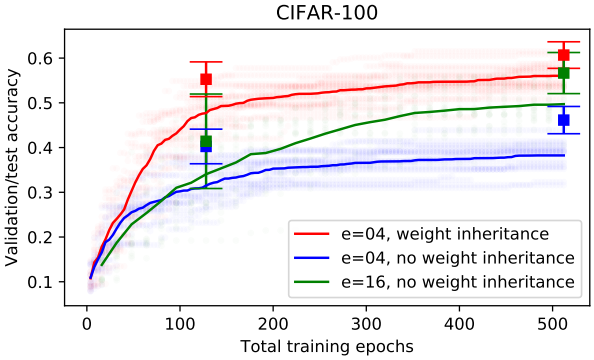

Figure 7.4: Training progression of the EA with weight inheritance compared to both baselines on CIFAR-100.

Table 7.2: Test accuracies at two checkpoints on CIFAR-100.

		128 total epochs			512 total epochs		
Inheritance	e	Min.	Mean	Max.	Min.	Mean	Max.
Yes	4	**47.7**	**55.3 ± 3.8**	**61.1**	**56.1**	**60.7 ± 2.9**	**66.1**
No	4	31.7	40.2 ± 3.8	46.0	39.8	46.1 ± 3.0	52.2
No	16	25.9	41.4 ± 10.3	57.7	46.6	56.7 ± 4.5	63.0

inheritance is used, minimum, mean, and maximum accuracies are higher than their baseline counterparts at all tested checkpoints.

For CIFAR-10, weight inheritance experiments reach a mean test accuracy of $85\,\% \pm 2\,\%$ after only 128 total training epochs. In comparison, baseline I experiments reach a mean test accuracy of $82\,\% \pm 1\,\%$ after 512 epochs. This means that the EA with weight inheritance achieves significantly $(p < 0.01)$ higher accuracy than baseline I in 75 % less total training epochs. Baseline II experiments reach a test accuracy of $85\,\% \pm 3\,\%$ after 512 epochs. This is slightly, though not significantly, lower than the test accuracy of the weight inheritance experiments. After 512 epochs, the weight inheritance experiments reach a mean test accuracy of $87\,\% \pm 2\,\%$, which is significantly $(p < 0.01)$ higher than baseline II at 512 epochs.

For CIFAR-100, results look very similar. After 128 epochs, the weight inheritance experiments achieve a mean test accuracy of $55\,\% \pm 4\,\%$. In contrast, baseline I experiments reach a significantly $(p < 0.01)$ lower mean test accuracy of $46\,\% \pm 3\,\%$ after 512 epochs. Again, this is an improvement using 75 % less total training epochs. Baseline II experiments achieve a (not significantly) higher mean test accuracy of $57\,\% \pm 5\,\%$ after 512 epochs. Running the weight inheritance experiments for all 512 epochs as well results in a mean test accuracy of $61\,\% \pm 3\,\%$, which now is significantly $(p < 0.01)$ higher than the baseline II test accuracy at 512 epochs.

Our best evolved network on CIFAR-100 without data augmentation reaches a test accuracy of 66.1 % after 512 total epochs and required 1×10^{16} FLOPs[1] to find.

[1]The FLOPs estimate for a single network is based on the FLOPs reported by the TensorFlow profiler to process a single example multiplied by 4 epochs, 98 batches per epoch and batch size 512. The

This takes about 1.5 days on a single NVIDIA K40 GPU and is a modest amount of computation, e.g. compared to Real et al. [98] who use 2×10^{20} FLOPs. In contrast to Real et al., none of our results reach state-of-the-art performance, but that was, as already pointed out, not the goal of this work.

In summary, weight inheritance experiments on CIFAR-10 and CIFAR-100 have shown to achieve significantly $(p < 0.01)$ higher accuracy using a quarter of the total training epochs when compared to baseline I that uses the same amount of training epochs per fitness evaluation. Furthermore, final accuracy after 512 epochs is also significantly $(p < 0.01)$ higher compared to baseline II experiments which benefited from more training epochs per fitness evaluation.

To get an idea how the evolutionary process modifies the genotypes, consider Figure 7.7. It shows how the genome length, i.e. the number of building blocks in the corresponding networks, changes over the course of evolution. All EA runs are initialized with a genotype that contains a single building block. During the evolutionary process, increasingly larger genotypes are evaluated as their phenotypes reach higher accuracy than those of genotypes with fewer building blocks. The weight inheritance experiments and baseline II both settle around an average of seven building blocks, whereas baseline I networks contain an average of six building blocks.

Additional experiments with 10 repetitions each have been performed on the smaller MNIST and Fashion-MNIST datasets. The results are shown in Figures 7.5 and 7.6. On both datasets, improvements from weight inheritance over its baselines are marginal. This is expected, as they are easy to solve compared to CIFAR-10 or CIFAR-100 and can be learned quickly by small networks. Still, there is no deterioration in performance from using weight inheritance either.

7.4.3 Discussion

The tradeoff between few and many training epochs per fitness evaluation that is explained in Section 7.4.1 has a visible effect in Figures 7.3 and 7.4. At the beginning of each experiment, baseline I outperforms baseline II, but at some point this relationship inverts. This is because small networks, which require only few epochs to reach good accuracy, are still sufficient to increase the validation set accuracy in the beginning of the experiment. However, eventually larger networks become necessary to further improve the results. These networks require more training time, making it easier for the algorithm that trains networks longer during fitness evaluation to progress. Therefore, the green and blue graphs intersect. This happens earlier for CIFAR-100, because it is a harder problem than CIFAR-10.

We see weight inheritance experiments consistently outperform their baselines on CIFAR-10 and CIFAR-100 but could not observe a significant difference on the

total FLOPs of the EA run is the sum of the FLOPs estimates for all networks that were trained during the optimization.

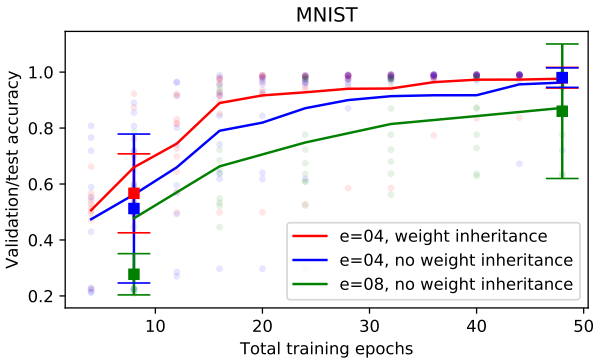

Figure 7.5: Training progression of the EA with weight inheritance compared to both baselines on MNIST.

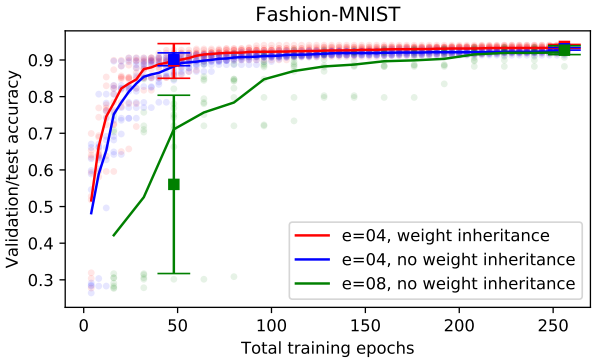

Figure 7.6: Training progression of the EA with weight inheritance compared to both baselines on Fashion-MNIST.

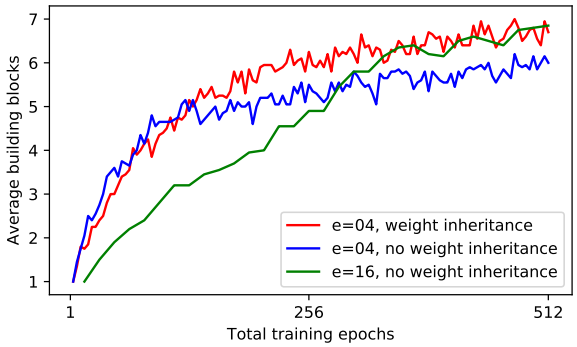

Figure 7.7: Average (of all EA runs) number of building blocks in the genome during the optimization process on CIFAR-100.

MNIST or Fashion-MNIST datasets. We do not find any instances in our experiments where weight inheritance is harmful, but this need not be the case generally. Just like in our work, most recent neuroevolution publications only use mutation operators and refrain from performing crossover. While this is usually motivated by the difficulty of designing a useful network crossover operator, crossover might also bring problems with regard to weight inheritance. Similar to choosing a bad initialization before starting the training of a network, building a new network from trained parts of different networks could leave it in a region of the weight space that is hard to optimize.

7.5 Conclusion

Evolutionary algorithms show promise as a way to automatically discover appropriate network architectures for new problems and tackle our first goal (G1). However, their usefulness is limited by their enormous computational requirements. Optimizing deep neural network architectures is computationally expensive because networks have to be retrained for each fitness evaluation. Therefore, approaches to lower these requirements are of great value.

We show that an evolutionary algorithm with a weight inheritance scheme generally achieves equal or higher accuracy compared to baselines that do not use weight inheritance and benefit from more training epochs per fitness evaluation. The fitness convergence speed is improved, sometimes making it possible to drastically reduce the number of total training epochs while achieving test accuracies comparable to the baselines. Specifically, on both CIFAR-10 and CIFAR-100 weight inheritance achieves similar or better accuracy than baselines in 75 % fewer training epochs. The resulting speedup makes evolutionary algorithms a lot more viable for application to neural network architecture optimization, even on hard problems, achieving our goal of reduced computational requirements (G2). If accuracy is more important than training time, weight inheritance can also lead to a higher final test accuracy in some cases. Most importantly though, we find no instance where weight inheritance is harmful. All results using our EA show either equally good or better results than the baselines.

However, despite reduced training times this approach to NAS is still expensive, as it requires the training of many networks. Research into one-shot NAS algorithms circumvents this problem by concurrently training a single set of weights and the architecture. We take a look at such an approach in the next chapter.

Chapter 8

Learned Weight Sharing for Deep Multi-Task Learning

After exploring EAs for weight training in Chapters 5 and 6, and for neural architecture search in Chapter 7, we now want to face the problem of creating an algorithm in the spirit of neuroevolution that optimizes both weights and architecture at once but apply it to large DNNs. In other words, we will work on a one-shot NAS algorithm. Previous chapters have shown that for weight training EAs are not yet competitive to SGD in supervised learning settings. On the other hand, they perform well for the architecture optimization which becomes costly overall due to the repeated SGD training processes. In this chapter, we will present an algorithm that combines an EA with SGD to learn weights and architecture and that does not need to train a network for each fitness evaluation.

We do so in the context of multi-task learning, i.e. multiple supervised tasks are learned simultaneously by a shared model. In deep multi-task learning, weights of task-specific networks are shared between tasks to improve performance on each single one. Since the question of which weights to share between layers is difficult to answer, human-designed architectures often share everything but a last task-specific layer. In many cases, this simplistic approach severely limits performance. Instead, we propose an algorithm to *learn* the assignment between a shared set of weights and task-specific layers. To optimize the non-differentiable assignment and at the same time train the differentiable weights, learning takes place via a combination of natural evolution strategy and stochastic gradient descent. The end result are task-specific networks that share weights but allow independent inference. They achieve lower test errors than baselines and methods from literature on three multi-task learning datasets.

The results presented in this chapter are based on the following publication with accompanying source code:

- Jonas Prellberg and Oliver Kramer. Learned weight sharing for deep multi-task learning by natural evolution strategy and stochastic gradient descent. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, UK, July 19-24, 2020*. IEEE, 2020
- https://github.com/jprellberg/learned-weight-sharing

Outline. Section 8.1 introduces the concept of deep multi-task learning and related work that implements deep MTL in various ways. The following Section 8.2 highlights the connection to NAS and presents the high-level idea of our learned weight sharing algorithm. In Section 8.3, the necessary background on natural evolution strategies is given. Section 8.4 explains our learned weight sharing method in detail and we showcase its performance on multiple datasets in Section 8.5. The chapter ends with a conclusion in Section 8.6.

8.1 Deep Multi-Task Learning

Deep learning systems have achieved remarkable success in various domains at the cost of massive amounts of labeled training data. This poses a problem in cases where such data is difficult or costly to acquire. In contrast, humans learn new tasks with minimal supervision by building upon previously acquired knowledge and reusing it for the new task. Transferring this ability to artificial learning is a long-standing goal that is being tackled from different angles [65, 25, 130]. A step in this direction is multi-task learning (MTL), which refers to learning multiple tasks at once with the intention to transfer and reuse knowledge between tasks in order to better solve each single task [14].

MTL is a general concept that can be applied to learning with different kinds of models. For the case of neural networks, MTL is implemented by sharing some amount of weights between task-specific networks (hard parameter sharing) or using additional loss functions or other constraints to create dependencies between otherwise independent weights of task-specific networks (soft parameter sharing). This way, better generalization may be achieved, because the network is biased to prefer solutions that apply to more than one task.

The main difference between various deep multi-task learning approaches is how weight sharing between tasks is implemented. This decision is encoded in the architecture of a deep neural network, either by the designer or an algorithm. Early works usually employ a shared neural network that branches into small task-specific parts at its end [153, 72]. This approach, referred to as full sharing in this chapter, is restrictive, because all tasks have to work on exactly the same representation, even if the tasks are very different. This motivates further work to lift this restriction and make weight sharing data-dependent.

Approaches like cross-stitch networks [81], sluice networks [104], or soft layer ordering [79] introduce additional differentiable parameters that control the weight sharing and are jointly optimized with the networks weights by SGD. In these approaches, the task-specific networks are connected by gates between every layer that perform weighted sums over the individual layer outputs. The coefficients of these weighted sums are learned and can therefore control the influence that different tasks have on each other. However, since all task-specific networks are interconnected, this

approach requires forward passes on all of them for a training step on any task and also when performing inference on only a single task. In contrast, every task-specific network created with our algorithm can perform forward passes independently. Soft layer ordering further has the restriction that all shareable layers at any position in the network must be compatible in input and output shape. Our approach, on the other hand, is unrestricted by the underlying network architecture and can, for example, be applied to standard residual networks.

Another set of works explores non-differentiable ways to share weights. Examples include fully-adaptive feature sharing, which iteratively builds a branching architecture that groups similar tasks together [73], or routing networks, which use reinforcement learning to choose a sequence of modules from a shared set of modules in a task-specific way [102]. Routing networks are similar to our work in that they also avoid the interconnection between task-specific networks, as described before. However, their approach also fundamentally differs from ours in that their routing network chooses layers in a data-dependent way on a per-example basis, while our network configuration is fixed after training and only differs between tasks not examples.

8.2 Neural Architecture Search for Deep Multi-Task Learning

In this chapter, we will perform deep MTL with hard parameter sharing and optimize how to share weights between tasks with a one-shot NAS algorithm. We confine ourselves to the common case where weights can only be shared between corresponding layers of each task-specific network. Figure 8.1 illustrates the resulting spectrum of possible sharing configurations with three possibilities to solve a three-task MTL problem. When disregarding the possibility to perform MTL, an independent set of weights is used in every task-specific network. We will refer to this configuration as no sharing and use it as our first baseline to check that the MTL approach actually performs better than single-task learning. A very simple form of MTL, sometimes called shared back-bone in literature but referred to as full sharing here, shares all weights but those of the final layer. This will form our second baseline to show improvements over a simple MTL approach. Our own approach is a middle-ground and learns a weight sharing configuration from data. The configuration shown in Figure 8.1 is a result on DKL-MNIST (see Section 8.5.1).

The difficulty now lies in choosing an appropriate weight sharing configuration from this extremely large search space. We use an automatic method that learns how to share layer weights between task-specific networks using alternating optimization with a natural evolution strategy (NES) and stochastic gradient descent. The main problem is the non-differentiable assignment between weights and layers that prevents learning both the assignment and weights with SGD. Therefore, we exploit the black-

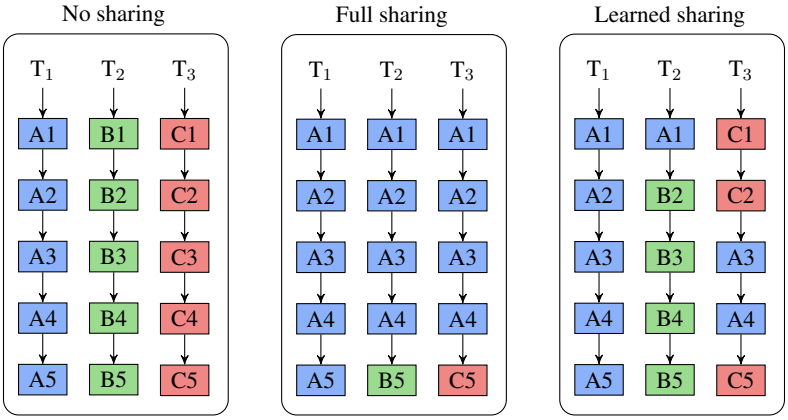

Figure 8.1: Different weight sharing schemes to solve a three-task MTL problem. For each scheme, three task-specific networks with five layers are shown. Layer weights are indicated by the letter in each layer and weights are shared between tasks if different task-specific networks use the same weight as indicated by the letter. Weights are never shared in the last layer because it is the task-specific output layer.

box nature of NES to optimize a probability distribution over the non-differentiable assignment. It would also be possible to learn the weights themselves with NES, but as we have seen in Chapter 5 disregarding the gradient is inefficient and should be avoided. Since for every fixed assignment the networks become differentiable w.r.t. their weights, we exploit SGD to efficiently train them.

While alternating these two steps, the probability distribution's entropy decreases, and the layer weights are optimized to perform well under the most likely assignments. In the end, this results in a single most likely assignment and corresponding layer weights. Notably, this is achieved without resorting to costly fitness evaluation steps that have to train networks from scratch or differentiable weight sharing approaches [81, 104, 79] that result in computationally intensive forward passes during inference.

This optimization process is a form of neural architecture search where a number of building blocks are predetermined, and the search process explores how to connect these building blocks. Different from typical NAS, there is not a single input and output but one for each task. By optimizing the structure of all task-specific networks together, they can share building blocks in their architectures and consequently share weights. It would also be possible to combine this setting with a search over different kinds of building blocks or a less restricted connection model that does not predetermine the overall architecture.

The algorithm that powers our learned weight sharing (LWS) approach is the aforementioned hybrid optimization of differentiable and non-differentiable parameters. It traces back to information-geometric optimization [87], which provides algorithms for the optimization of arbitrary probability distributions using natural gradient methods, and it has been combined with SGD for problems that are not completely black-box.

Shirakawa et al. [114] perform a simple form of NAS that is based on binary decisions. These non-differentiable decisions are modeled with Bernoulli distributions and optimized using NES. Shirakawa et al. then show that it is possible to simultaneously train the differentiable network weights with SGD by using a Monte-Carlo approximation of the weight gradient.

Akimoto et al. [2] build on this algorithm and apply it to a more general neural architecture search space that allows to select between different kinds of building blocks for (single-task) image classification and image inpainting. These non-differentiable choices are modeled with a categorical distribution and optimized by NES, while SGD optimizes the network weights as before.

Lenc et al. [67] also use the hybrid optimization algorithm to simultaneously optimize non-differentiable binary sparsity masks via a Bernoulli distribution and differentiable layer weights. This allows to train sparse networks directly instead of sparsifying them after first training a dense model, which would require vastly more memory and computation.

We model deep MTL as an assignment problem between layer weights and task-specific networks that is expressed with categorical distributions. This allows us to use the same concept of hybrid NES and SGD optimization to efficiently search for a MTL architecture in the large combinatorial search space, while training the layer weights at the same time. The resulting MTL architectures yield accuracy improvements compared to our own baselines and baselines from literature on three datasets.

8.3 Natural Evolution Strategy

After working with population-based EAs, we now shift the focus to a different kind of EA. Natural evolution strategy refers to a class of black-box optimization algorithms that update a search distribution in the direction of higher expected fitness using the natural gradient [142]. In contrast to population-based EAs, there is no actual population of solutions, but instead the information that would be contained in the population is encoded as a probability distribution. Again, we consider an optimization problem $\max_x u(x)$ over an arbitrary solution space $x \in \mathcal{X}$ with a black-box fitness function $u(x)$.

Let us first discuss the concept of a search gradient. A search distribution over the solution space $\mathcal{X}$ with probability density function $q(x|\alpha)$ is defined. It is parameterized by a real-valued vector α. When sampling and evaluating solutions from such a distribution the expected fitness is given by

$$J(\alpha) = \mathbb{E}_{q_\alpha}[u(x)]. \tag{8.1}$$

We are interested in the gradient $\nabla_\alpha J(\alpha)$ in order to change the distribution's param-

eters in the direction of higher expected fitness. It is given by [142] as

$$\begin{aligned}
\nabla_\alpha J(\alpha) &= \nabla_\alpha \int u(x)\, q(x|\alpha)\, \mathrm{d}x && (8.2)\\
&= \int u(x)\, \nabla_\alpha q(x|\alpha)\, \mathrm{d}x && (8.3)\\
&= \int u(x)\, \frac{q(x|\alpha)}{q(x|\alpha)} \nabla_\alpha q(x|\alpha)\, \mathrm{d}x && (8.4)\\
&= \int u(x)\, q(x|\alpha)\, \frac{\nabla_\alpha q(x|\alpha)}{q(x|\alpha)} \mathrm{d}x && (8.5)\\
&= \int u(x)\, q(x|\alpha)\, \nabla_\alpha \log q(x|\alpha)\, \mathrm{d}x && (8.6)\\
&= \mathbb{E}_{q_\alpha}\left[u(x)\, \nabla_\alpha \log q(x|\alpha)\right], && (8.7)
\end{aligned}$$

where they use the so-called log-likelihood trick to arrive at Equation 8.6. This trick simply consists of using the relationship

$$\nabla_\alpha \log q(x|\alpha) = \frac{\nabla_\alpha q(x|\alpha)}{q(x|\alpha)}, \tag{8.8}$$

which follows from calculus rules about the derivative of the logarithm. It is now possible to estimate the search gradient in the direction of higher expected fitness for any search distribution for that we can calculate the log-derivative $\nabla_\alpha \log q(x|\alpha)$.

Since the gradient $\nabla_\alpha J(\alpha)$ is expressed as an expectation in Equation 8.7, it can easily be approximated from samples $x_1, \ldots, x_\lambda$ distributed according to $q(x|\alpha)$ by the Monte-Carlo estimate

$$\nabla_\alpha J(\alpha) \approx \frac{1}{\lambda} \sum_{i=1}^{\lambda} u(x_i)\, \nabla_\alpha \log q(x_i|\alpha), \tag{8.9}$$

where λ is the number of samples. Using this gradient, the distribution parameters α can be adjusted by taking a small step in its direction. This search gradient evolution strategy as outline by [142] is summarized in Algorithm 5.

Instead of following the plain gradient directly, NES follows the natural gradient $\mathbf{F}^{-1}\nabla_\alpha J(\alpha)$. Here, $\mathbf{F}^{-1}$ refers to the inverse of the search distribution's Fisher information matrix. Gradient descent using the natural gradient ensures that the Kullback–Leibler divergence between the search distribution $q(x|\alpha)$ before and after the update is of a constant small stepsize, i.e. the distribution is only slightly modified after taking a single step. In contrast, following the plain gradient might drastically alter the distribution, e.g. due to outliers in the current set of samples used to calculate the gradient.

The Fisher information matrix depends only on the probability distribution itself and can often be analytically derived, e.g. for the common case of multinormal search distributions [127]. For the case of search distributions from the exponential family

Algorithm 5: Search gradient evolution strategy, see [142].

Let $q(x|\alpha)$ be the search distribution
while termination condition not met **do**
  for i in $1 \dots \lambda$ **do**
    sample x_i distributed according to $q(x|\alpha)$
    calc. fitness $u_i = u(x_i)$
    calc. log-derivative $\nabla_\alpha \log q(x_i|\alpha)$
  end
  $\nabla_\alpha J(\alpha) = \frac{1}{\lambda} \sum_{i=1}^{\lambda} u_i \nabla_\alpha \log p(x_i|\alpha)$
  $\alpha \leftarrow \alpha + \eta \nabla_\alpha J(\alpha)$
end

under expectation parameters, a direct derivation of the natural gradient without first calculating $\mathbf{F}$ is given by [87, page 57] and will be presented in the remainder of this section. The probability density function for members of the exponential family has the form

$$q(x|\alpha) = h(x) \exp(\alpha \cdot T(x) - A(\alpha)) \tag{8.10}$$

with natural parameter vector α, sufficient statistic vector $T(x)$ and cumulant function $A(\alpha)$. We focus only on the case where $h(x) = 1$ in our work. If we reparameterize the distribution with a parameter vector μ that satisfies

$$\mu = \mathbb{E}_{q_\alpha}[T(x)] = \nabla_\alpha A(\alpha), \tag{8.11}$$

then we call μ the expectation parameters. With such a parameterization, there is a nice result regarding the natural gradient: The natural gradient w.r.t. the expectation parameters is given by the plain gradient w.r.t. the natural parameters, i.e.

$$\tilde{\nabla}_\mu q(x|\mu) = \nabla_\alpha q(x|\alpha). \tag{8.12}$$

It follows that the log-derivative, which is necessary for the search gradient estimate in NES, can easily be derived as

$$\begin{aligned} \tilde{\nabla}_\mu \log q(x|\mu) &= \nabla_\alpha \log q(x|\alpha) && (8.13)\\ &= \nabla_\alpha (\alpha \cdot T(x) - A(\alpha)) && (8.14)\\ &= T(x) - \mu && (8.15) \end{aligned}$$

because of the relationship in Equation 8.11 between gradient of the cumulant function and expectation parameters. In other words, if we choose a search distribution with expectation parameters, the plain and natural gradient coincide. We will use this fact later, to follow the natural gradient of a categorical distribution.

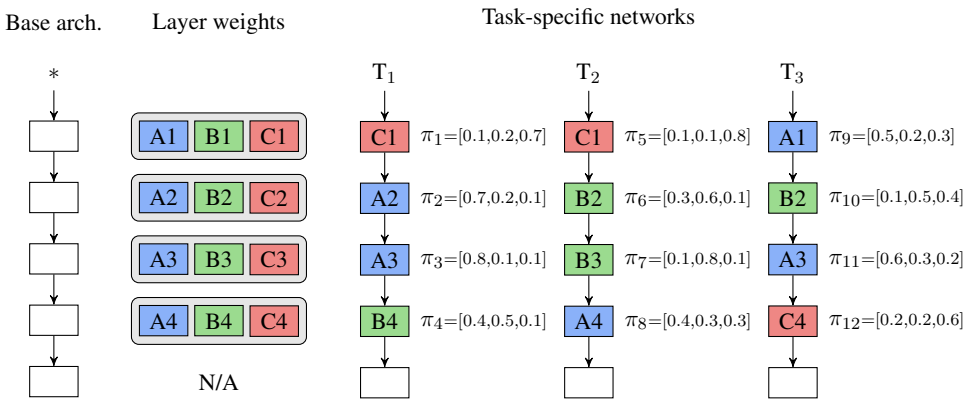

Figure 8.2: Setup to solve a three-task MTL problem with LWS. A base architecture is duplicated for each task and weights are stochastically assigned to each layer. There are a total of $N = 12$ layers and $K = 3$ weights per weight set. The depicted assignment is the most probable one, which is used for inference.

8.4 Learned Weight Sharing

Consider the setup depicted in Figure 8.2 to solve an MTL problem using deep neural networks. Any neural network architecture is chosen as the base architecture, e.g. a residual network. This base architecture is duplicated once for each task to create task-specific networks. Finally, the last layer of each task-specific network is modified to have the appropriate number of outputs for the task.

In this setup, the weights of every layer except for the last one are compatible between task-specific networks and can potentially be shared. To this end, a set of K weights is created for every layer, and all of the N task-specific network layers are assigned a weight from their corresponding set. By assigning the same weight to multiple task-specific networks, weight sharing is achieved. The number of weights per layer must not necessarily be the same, however, we restrict ourselves to equally sized sets of weights for simplicity.

The problem is now to find good assignments between the weights and task-specific network layers and, at the same time, train the weights themselves. We achieve this by alternating between the optimization of a search distribution over assignments with NES and the optimization of layer weights with SGD. This approach is summarized in Algorithm 6 and explained in more detail below.

8.4.1 Learning Objective

Since the hybrid optimization algorithm is fundamentally the same algorithm as in [114, 2, 67], we choose to frame the learning objective in the same way as [2] because of their clear mathematical approach. The search for good assignments and layer

Algorithm 6: Learned weight sharing training procedure.

Let $p(a|\pi)$ be the search distribution over assignments
Let $\mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta, a)$ be the loss for a batch of data $\mathbf{x}, \mathbf{y}$ under weights θ and assignment a
def `StepNES` (θ, π) **as**
    $\mathbf{x}, \mathbf{y} \leftarrow$ get random batch of training data
    for i in $1 \dots \lambda_\pi$ **do**
        sample a_i distributed according to $p(a|\pi)$
        calc. loss $l_i = \mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta, a_i)$
        calc. log-derivative $\nabla_\pi \log p(a_i|\pi)$
    end
    calc. utilities $u_i = 2 \cdot \frac{\text{rank}(l_i)-1}{\lambda_\pi - 1} - 1$
    $\nabla_\pi J_\pi = \frac{1}{\lambda_\pi} \sum_{i=1}^{\lambda_\pi} u_i \nabla_\pi \log p(a_i|\pi)$
    return $\pi + \eta_\pi \nabla_\pi J_\pi$
end
def `StepSGD` (θ, π) **as**
    $\mathbf{x}, \mathbf{y} \leftarrow$ get random batch of training data
    for i in $1 \dots \lambda_\theta$ **do**
        sample a_i distributed according to $p(a|\pi)$
        calc. weight gradient $\nabla_\theta \mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta, a_i)$
    end
    $\nabla_\theta J_\theta = \frac{1}{\lambda_\theta} \sum_{i=1}^{\lambda_\theta} \nabla_\theta \mathcal{L}_{\mathbf{x},\mathbf{y}}(\theta, a_i)$
    return $\theta - \eta_\theta \nabla_\theta J_\theta$
end
$\pi \leftarrow$ search distribution parameter vector with all $N(K-1)$ elements set to $\frac{1}{K}$
$\theta \leftarrow$ randomly initialized neural network weights
while not finished **do**
    $\pi \leftarrow$ `StepNES` (θ, π)
    $\theta \leftarrow$ `StepSGD` (θ, π)
end

weights is cast as an optimization problem

$$\min_{\theta,a} \mathcal{L}(\theta, a), \tag{8.16}$$

where $\mathcal{L} : \Theta \times \mathcal{A} \to \mathbb{R}$ is the average loss over all tasks, $\theta \in \Theta$ is a vector of all layer weights, and $a \in \mathcal{A}$ is an assignment of weights to task-specific network layers. The loss function $\mathcal{L}$ is differentiable w.r.t. θ but black-box w.r.t. a. We would like to exploit the fact that θ can be efficiently optimized by SGD but need a way to simultaneously optimize a. Therefore, we create a stochastic version of the problem

$$\min_{\theta,\pi} J(\theta, \pi) = \mathbb{E}_{p_\pi}[\mathcal{L}(\theta, a)] \tag{8.17}$$

by introducing a probability distribution defined on $\mathcal{A}$ with density function $p(a|\pi)$. This stochastic formulation makes the assignments amenable for optimization through π by the NES algorithm but, on the other hand, requires to sample assignments for the calculation of the gradient w.r.t. θ.

8.4.2 Assignment Optimization

We use the NES algorithm to optimize π for lower expected loss $J(\theta, \pi)$ while keeping θ fixed (see Algorithm 6, lines 3 to 13). The parameter π is initialized so that all assignments are equally probable, but with prior knowledge the initial parameter vector could also be chosen so that it is biased towards certain preferred assignments.

Assignments $a_1, \ldots, a_{\lambda_\pi}$ distributed according to $p(a|\pi)$ are sampled and their loss values $l_i = \mathcal{L}(\theta, a_i)$ are calculated on the same batch of training data for every assignment. Following Equation 8.9, the search gradient can be approximated as

$$\nabla_\pi J(\theta, \pi) \approx \frac{1}{\lambda_\pi} \sum_{i=1}^{\lambda_\pi} u_i \nabla_\pi \log p(a_i|\pi) \tag{8.18}$$

with utility values u_i in place of fitness values (see below). Finally, π is updated by performing a step in the direction of $\nabla_\pi J(\theta, \pi)$ scaled by a learning rate parameter η_π. This is basic SGD but in principle more sophisticated optimizers like SGD with momentum or Adam could be used for this update step as well.

The utility values are created by fitness shaping to make the algorithm invariant to the scale of the loss function. Loss values l_i are transformed into utility values

$$u_i = 2 \cdot \frac{\text{rank}(l_i) - 1}{\lambda_\pi - 1} - 1, \tag{8.19}$$

where rank (l_i) ranks the loss values from 1 to λ_π in descending order, i.e. the smallest l_i receives rank λ_π. This results in equally spaced utility values in $[-1, 1]$ with the lowest loss value receiving a utility value of one.

8.4.3 Layer Weight Optimization

While we can use backpropagation to efficiently determine the weight gradient $\nabla_\theta \mathcal{L}(\theta, a)$ with a fixed, determining $\nabla_\theta J(\theta, \pi)$ with π fixed on the stochastic problem version is not possible directly. Instead, we use a Monte-Carlo approximation to optimize θ for lower expected loss $J(\theta, \pi)$ while keeping π fixed (see Algorithm 6, lines 14 to 22). This Monte-Carlo approximation as part of a hybrid optimization framework has first been described by Shirakawa et al. [114].

In the beginning, all layer weights θ are randomly initialized. For the Monte-Carlo gradient estimation, assignments $a_1, \ldots, a_{\lambda_\theta}$ distributed according to $p(a|\,\pi)$ are sampled, and backpropagation is performed for each sample. The same batch of training data is used for the backpropagation step throughout this process for every assignment. The resulting gradients $\nabla_\theta \mathcal{L}(\theta, a_i)$ are averaged over all assignments, so that the final gradient is given by

$$\nabla_\theta J(\theta, \pi) \approx \frac{1}{\lambda_\theta} \sum_{i=1}^{\lambda_\theta} \nabla_\theta \mathcal{L}(\theta, a_i)\,. \tag{8.20}$$

Using this gradient, θ is updated by SGD with learning rate η_θ but, again, more sophisticated optimizers could be employed instead.

8.4.4 Natural Gradient

The NES search gradient calculation in Equation 8.18 actually follows the plain gradient instead of the natural gradient unless we take care to use a specific parameterization for the search distribution. As previously explained, the natural gradient and plain gradient coincide when the distribution is a member of the exponential family and has expectation parameters. In our problem setting, there are a total of N layers distributed over all task-specific networks that need to be assigned a weight from K possible choices from the weight set corresponding to each layer. We can model this with categorical distributions, which are part of the exponential family, as follows.

First, consider a categorical distribution over K categories with samples $x \in [\![K]\!]$. It is well known [29] that the categorical distribution can be written in exponential family form (see Equation 8.10) with natural parameters $\alpha \in \mathbb{R}^{K-1}$ as

$$p_{\text{nat}}(x|\,\alpha) = \exp\left(\alpha \cdot T_{\text{nat}}(x) - A_{\text{nat}}(\alpha)\right) \tag{8.21}$$

$$T_{\text{nat}}(x) = \begin{pmatrix} \delta_{1,x} & \cdots & \delta_{K-1,x} \end{pmatrix} \tag{8.22}$$

$$A_{\text{nat}}(\alpha) = \log\left(1 + \sum_{i=1}^{K-1} e^{\alpha_i}\right), \tag{8.23}$$

where $\delta_{i,j}$ is the Kronecker delta function that is 1 if $i = j$ and 0 otherwise. Our goal is to have this distribution in expectation parameters so that we can use the results

mentioned before for the natural gradient calculation. We can reparameterize the distribution as

$$p_{\text{ex}}(x|\,\mu) = \exp\left(r_{\text{ex}}(\mu) \cdot T_{\text{ex}}(x) - A_{\text{ex}}(\mu)\right) \tag{8.24}$$

$$r_{\text{ex}}(\mu) = \begin{pmatrix} \log\frac{\mu_1}{\mu_K} & \cdots & \log\frac{\mu_{K-1}}{\mu_K} \end{pmatrix} \tag{8.25}$$

$$T_{\text{ex}}(x) = \begin{pmatrix} \delta_{1,x} & \cdots & \delta_{K-1,x} \end{pmatrix} \tag{8.26}$$

$$A_{\text{ex}}(\mu) = -\log\mu_K, \tag{8.27}$$

which gives us a parameter vector $\mu \in [0,1]^{K-1}$ with entries corresponding to the probabilities of all but the last category. For notational convenience, we use $\mu_K = 1 - \sum_{i=1}^{K-1}\mu_i$ even though μ_K is not technically part of the parameter vector.

First, to see why Equation 8.27 is equal to Equation 8.23 under the reparameterization, consider the following steps:

$$A_{\text{ex}}(\mu) = A_{\text{nat}}(r_{\text{ex}}(\mu)) \tag{8.28}$$

$$= \log\left(1 + \sum_{i=1}^{K-1}\frac{\mu_i}{\mu_K}\right) = \log\left(1 + \frac{1}{\mu_K}\sum_{i=1}^{K-1}\mu_i\right) \tag{8.29}$$

$$= \log\left(1 + \frac{1}{\mu_K}(1-\mu_K)\right) = \log\left(\frac{1}{\mu_K}\right) \tag{8.30}$$

$$= -\log\mu_K. \tag{8.31}$$

Then, to see that μ are expectation parameters, we compare it to the derivative of the cumulant function in natural parameters (see Equation 8.11). By using the relationship $\alpha_i = \log\frac{\mu_i}{\mu_K}$ between natural parameters and the reparameterization, we can see that they are equal for all $i \in [\![K-1]\!]$:

$$\frac{\partial A_{\text{nat}}(\alpha)}{\partial\alpha_i} = \frac{e^{\alpha_i}}{1+\sum_{j=1}^{K-1}e^{\alpha_j}} = \frac{\frac{\mu_i}{\mu_K}}{1+\sum_{j=1}^{K-1}\frac{\mu_j}{\mu_K}} \tag{8.32}$$

$$= \frac{\mu_i}{\mu_K + \sum_{j=1}^{K-1}\mu_j} = \frac{\mu_i}{\mu_K + 1 - \mu_K} \tag{8.33}$$

$$= \mu_i. \tag{8.34}$$

Now that we have a categorical distribution with expectation parameters, consider a joint of N independent but *not* identically distributed categorical distributions with samples a and parameters π so that

$$a = \begin{pmatrix} a_1 & \cdots & a_N \end{pmatrix} \in [\![K]\!]^N \tag{8.35}$$

$$\pi = \begin{pmatrix} \pi_1 & \cdots & \pi_N \end{pmatrix} \in [0,1]^{N(K-1)} \tag{8.36}$$

are the concatenations of the samples and expectation parameters of all N categorical

distributions, i.e. π is the concatenation of N parameter vectors $\pi_i \in [0,1]^{K-1}$.

Due to the independence of the N categorical distributions, the density function for the joint distribution becomes the product of their individual densities. Again, this is a member of the exponential family with expectation parameters:

$$p(a|\pi) = \prod_{i=1}^{N} p_{\text{ex}}(a_i|\pi_i) \tag{8.37}$$

$$= \prod_{i=1}^{N} \exp\left(r_{\text{ex}}(\pi_i) \cdot T_{\text{ex}}(a_i) - A_{\text{ex}}(\pi_i)\right) \tag{8.38}$$

$$= \exp\left(\sum_{i=1}^{N} r_{\text{ex}}(\pi_i) \cdot T_{\text{ex}}(a_i) - \sum_{i=1}^{N} A_{\text{ex}}(\pi_i)\right) \tag{8.39}$$

$$= \exp\left(r(\pi) \cdot T(a) - A(\pi)\right) \tag{8.40}$$

$$r(\pi) = \begin{pmatrix} r_{\text{ex}}(\pi_1) & \cdots & r_{\text{ex}}(\pi_N) \end{pmatrix} \tag{8.41}$$

$$T(a) = \begin{pmatrix} T_{\text{ex}}(a_1) & \cdots & T_{\text{ex}}(a_N) \end{pmatrix} \tag{8.42}$$

$$A(\pi) = \sum_{i=1}^{N} A_{\text{ex}}(\pi_i). \tag{8.43}$$

Equation 8.40 follows from the previous line, because the dot product between two vectors can just as well be computed over individual aligned parts of the vector that are later summed.

In summary, LWS uses $p(a|\pi)$ from Equation 8.37 as the density for its search distribution. The parameters π are the concatenation of all but the last probabilities for each categorical distribution. Since π are expectation parameters, we can use Equation 8.15 to calculate the natural gradient as

$$\nabla_\pi \log p(a|\pi) = T(a) - \pi \tag{8.44}$$

and plug it into Algorithm 6 at line 8.

8.4.5 Inference

After training has finished, the most likely weight assignment $\arg\max_a p(a|\pi)$ is used for inference. Given this assignment, a DNN can be constructed for each task, and they can be independently applied to input data from the respective task.

8.5 Experiments

We demonstrate the performance of LWS on three different multi-task datasets using convolutional network architectures taken from other MTL publications to compare our results to theirs. We also perform experiments using a residual network [39]

Table 8.1: Test error of learned weight sharing compared to full sharing and no sharing baselines on three datasets.

Method	DKL-MNIST ConvNet	CIFAR-100 ResNet18	Omniglot ResNet18
Full sharing	14.16 ± 0.37	31.80 ± 0.44	10.97 ± 0.60
No sharing	12.80 ± 0.16	32.53 ± 0.32	15.82 ± 1.02
Learned sharing	$\mathbf{11.83 \pm 0.51}$	$\mathbf{30.84 \pm 0.49}$	$\mathbf{10.70 \pm 0.62}$

architecture to show applicability of LWS to modern architectures. Furthermore, we provide two baseline results for all experiments which are full sharing, i.e. every task shares weights with every other task at each layer except for the last one, and no sharing, i.e. all task-specific networks are completely independent. Note that a completely independent network for each task means that its whole capacity is available to learn a single task, whereas the full sharing network has to learn all tasks using the same capacity. Depending on network capacity, task difficulty, and task compatibility, we will see no sharing outperform full sharing and vice versa.

All experiments are repeated 10 times and reported with mean and standard deviation. For statistical significance tests, we perform a one-sided Mann-Whitney U test. The search distribution parameters π are initialized to $\frac{1}{K}$ so that layers are chosen uniformly at random in the beginning. Furthermore, to prevent that a layer will never be chosen again once its probability reaches zero, every entry in π is clamped above $0.1\,\%$ after the update step and then π is renormalized to sum to one. The layer weights θ are initialized with He-uniform [38] initialization and update steps on θ are performed with the Adam [59] optimizer. All batches are created by sampling 16 training examples from each different task and concatenating them. Full sharing, no sharing, and LWS all use the same equal loss weighting between different tasks. MTL is usually sensitive to this weighting and further improvements might be achieved, but its optimization is left for future work.

8.5.1 DKL-MNIST

DKL-MNIST is a custom MTL dataset created from the Extended-MNIST [18] and Kuzushiji-MNIST [17] image classification datasets. The classification of digits, letters, and Japanese kuzushiji characters are different but related tasks, which provides good conditions to perform MTL. We select 500 training examples of digits, letters, and kuzushiji each for a total of 1,500 training examples and keep the complete test sets for a total of 70,800 test examples. Using only a few training examples per task creates a situation where sharing features between tasks should improve performance. Since all three underlying datasets are MNIST variants, the training examples are 28×28 pixel grayscale images, but there are 10 digit classes, 26 letter classes, and 10 kuzushiji classes in each task respectively.

For this small dataset, we use a custom convolutional network architecture that consists of three convolutional layers and two dense layers. The convolutions all have 32 output channels, kernel size 3×3, and are followed by batch normalization, ReLU activation, and 2×2 max-pooling. The first dense layer has 128 units and is followed by a ReLU activation, while the second dense layer has as many units as there are classes for the task. LWS is applied to the three convolutional layers and the first dense layer, i.e. the whole network except for the task-specific last layer.

We train LWS and the two baselines for 5,000 iterations on DKL-MNIST using a SGD learning rate of $\eta_\theta = 10^{-3}$. Furthermore, LWS uses $\lambda_\theta = \lambda_\pi = 8$ samples for both SGD and NES, and a NES learning rate of $\eta_\pi = 10^{-2}$ to learn to share sets of $K = 3$ weights for each layer. We see in Table 8.1 that full sharing at 14.16 % test error performs worse than no sharing at 12.80 % test error, i.e. there is negative transfer when using the simple approach of sharing all but the last layer. However, using LWS we find an assignment that is significantly ($p < 0.01$) better than the no sharing baseline for a total error of 11.83 %.

8.5.2 CIFAR-100

CIFAR-100 is a popular image classification dataset that contains 50,000 training examples and 10,000 test examples, all of which are 32×32 pixel RGB images. We cast it as an MTL problem by grouping the different classes into tasks by the 20 coarse labels that CIFAR-100 provides. For example, from the coarse label "flowers" a classification task with the five classes orchids, poppies, roses, sunflowers, and tulips is created. Each task then contains 5 classes and a total of 2,500 training examples (500 per class).

We employ the neural network architecture given by [102] to allow for a comparison against their results. It consists of four convolutional layers and four dense layers. The convolutions all have 32 output channels, kernel size 3×3, and are followed by batch normalization, ReLU activation, and 2×2 max-pooling. The first three dense layers all have 128 units and are followed by a ReLU activation, while the last dense layer has as many units as there are classes for the task, i.e. 5 for all tasks on this dataset. In [102] the authors only apply their MTL method to the three dense layers with 128 units, so we do the same for a fair comparison. This means the convolutional layers always share their weights between all tasks.

We train LWS and the two baselines for 4,000 iterations on CIFAR-100 using a SGD learning rate of $\eta_\theta = 10^{-3}$. Furthermore, LWS uses $\lambda_\theta = \lambda_\pi = 8$ samples for SGD and NES, and a NES learning rate of $\eta_\pi = 10^{-1}$ to learn to share sets of $K = 20$ weights for each layer. Table 8.2 shows that LWS, with a test error of 37.43 %, outperforms both cross-stitch networks at 47 % test error and routing networks at 40 % test error. However, no sharing achieves even better results. This can be attributed to the network capacity being small in relation to the dataset difficulty. In this case,

Table 8.2: Comparison against results from [102] using their network architecture.

Method	CIFAR-100 Test Error [%]
Cross-stitch networks [102]	47
Routing networks [102]	40
Full sharing	39.08 ± 0.36
No sharing	$\mathbf{36.50 \pm 0.43}$
Learned sharing	37.43 ± 0.53

having 20 times more weights is more important than sharing data between tasks.

Therefore, we repeat the experiment with a ResNet18 architecture that has much higher capacity than the custom convolutional network from [102]. The channel configuration in our ResNet18 is the same as in the original publication about residual networks [39]. However, due to the much smaller image size of CIFAR-100, we remove the 3×3 max-pooling layer and set the convolutional stride parameters so that downsampling is only performed in the last three stages. We apply LWS to share weights between each residual block. They are treated as a single unit that consists of two convolutional layers and, in the case of a downsampling block, a third convolutional layer in the shortcut connection. All hyperparameters stay the same except for the amount of iterations, which is increased to 20,000.

Test curves are shown in Figure 8.3 and final test results are listed in Table 8.1. We notice that no sharing at 32.53 % test error now performs worse than full sharing at 31.80 % test error. We believe the reason to be the increased network capacity that is now high enough to benefit from data sharing between tasks. LWS further improves on this and achieves the lowest test error at 30.84 %, which is significantly ($p < 0.01$) better than full sharing.

Depending on the sharing configuration, the total number of weights that are present in the system comprised of all task-specific networks differs. Naturally, the no sharing configuration has the highest possible amount of weights at 223M, while full sharing has the lowest possible amount at 11M. They differ exactly by a factor of 20, which is the number of tasks in this setting. LWS finds a configuration that uses 136M weights while still achieving higher accuracy than both baselines.

8.5.3 Omniglot

The Omniglot dataset [64] is a standard MTL dataset that consists of handwritten characters from 50 different alphabets, each of which poses a character classification task. The alphabets contain varying numbers of characters, i.e. classes, with 20 grayscale example images of 105×105 pixels each. Since Omniglot contains no predefined train-test-split, we randomly split off 20 % as test examples from each alphabet.

We employ the neural network architecture given by [79] to allow for a comparison

Table 8.3: Comparison against results from [79] using their network architecture.

Method	Omniglot Test Error [%]
Soft layer ordering [79]	24.1
Full sharing	20.85 ± 1.07
No sharing	23.52 ± 1.25
Learned sharing	$\mathbf{19.31 \pm 2.54}$

against their results. It consists of four convolutional layers and a single dense layer. The convolutional layers all have 53 output channels, kernel size 3×3, and are followed by batch normalization, ReLU activation, and 2×2 max-pooling. The final dense layer has as many units as there are classes for the task. As in [79], we apply LWS to the four convolutional layers.

We train LWS and the two baselines for 20,000 iterations on Omniglot using a SGD learning rate of $\eta_\theta = 10^{-3}$. Furthermore, LWS uses $\lambda_\theta = \lambda_\pi = 8$ samples for SGD and NES, and a NES learning rate of $\eta_\pi = 10^{-2}$ to learn to share sets of $K = 20$ weights for each layer. Table 8.3 shows how LWS outperforms SLO and both baselines. We repeat the experiment with a ResNet18 architecture with the 3×3 max-pooling removed and present results in Table 8.1. LWS still performs significantly ($p < 0.01$) better than no sharing and is on par with full sharing. Neither full sharing nor LWS is significantly ($p < 0.01$) better than the other.

8.5.4 Qualitative Results

Figure 8.4 sheds light on what kind of assignments are learned on the DKL-MNIST dataset. The three convolutional layers and the one dense layers that are shareable are denoted on the horizontal axis in the same order as in the network itself. For each layer, a stacked bar represents the percentage of tasks over all repetitions that shared the layer weight within a group of t tasks. Since there are three tasks and three weights per shared set, the only possible assignments are (1) all three tasks have independent weights, (2) two tasks share the same weight, while the last task has an independent weight, and (3) all three tasks share the same weight. In Figure 8.4, the group sizes correspond to these three assignments, e.g. in 40 % of the experiments the first layer had three tasks with independent weights. An exemplary assignment that was found in one of the DKL-MNIST experiments can be seen in Figure 8.1.

Figure 8.5 shows the same kind of visualization on Omniglot for a ResNet18. Due to the vastly increased number of possible assignments, the interpretation is not as straightforward as in the DKL-MNIST case. However, we can clearly see how weights are shared between a larger number of tasks in the early layers. This corresponds well to results from transfer learning literature [148], where early convolutional layers have been found to learn very general filters.

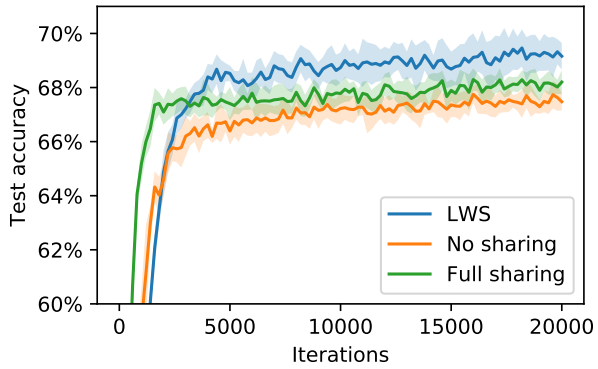

Figure 8.3: Test accuracy during training for LWS and its baselines using a ResNet18 on CIFAR-100.

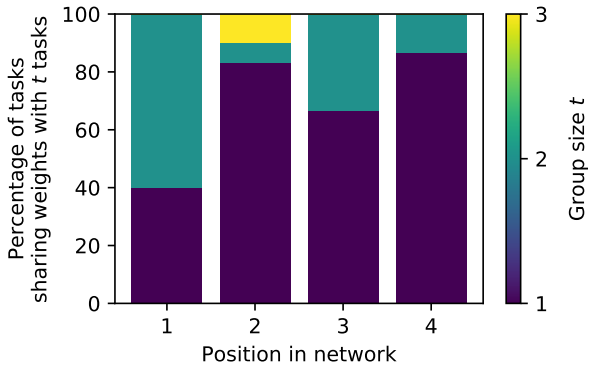

Figure 8.4: Percentage of tasks that share weights between exactly t tasks when learned with LWS on DKL-MNIST.

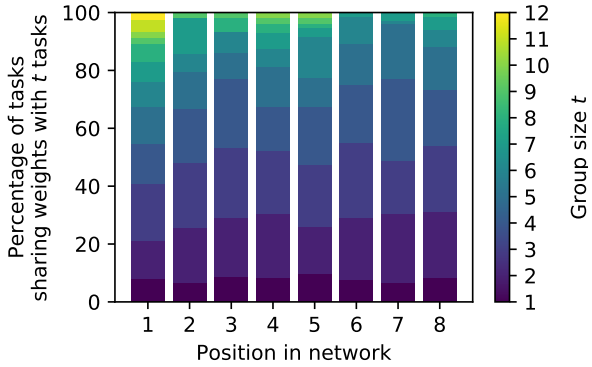

Figure 8.5: Percentage of tasks that share weights between exactly t tasks when learned with LWS on Omniglot.

8.6 Conclusion

LWS solves MTL problems by learning how to share weights between task-specific networks. We show how combining NES and SGD creates a learning algorithm that deals with the problem's non-differentiable structure by NES, while still exploiting the parts that *are* differentiable with SGD. This approach beats the MTL approaches cross-stitch networks, routing networks, and soft layer ordering on their respective problems, and we show good performance on three datasets using a large-scale residual network.

Through the use of MTL, this approach can be helpful in settings where labeled datasets are small but other similar datasets are available. Usually, it would be difficult to use them, because they are labeled differently or have a different format, but these deviations are acceptable in an MTL setting. The small size of the target dataset is offset through the MTL procedure, achieving the goal of increased data efficiency (G3). The NAS approach to the configuration of weight sharing for deep MTL minimizes the amount of manual decisions that have to be taken. This reduces the expertise necessary to apply MTL in practice and is in line with the motivations of our goal to create NAS algorithms (G1). Furthermore, LWS is efficient compared to other NAS approaches as presented in Chapter 7, because it is a one-shot algorithm, i.e. trains weights and architecture in a single pass. This tackles our goal of computational efficiency (G2).

Chapter 9

Conclusion

Deep learning has emerged as a powerful machine learning technique that is able to learn and generalize complex functions from data. This predictive capability is useful in an immense number of different real-world applications, e.g. supporting medical professionals in various ways [48, 90, 91, 35, 133, 57, 3, 37]. Many of these tasks require to extract information from images, which is what we focus on in this thesis. While it is getting easier and easier for everyone to apply deep learning to their own problems, it is difficult to achieve optimal results without machine learning expertise due to the immense number of hyperparameters that influence the training result. Even machine learning experts have to rely on experience, because it is impossible to exhaustively search for the best configuration by hand.

However, the burden on the machine learning practitioner can be eased through automated hyperparameter tuning and by extension automated machine learning. Initially, the development of such techniques just shifts the necessary expertise from the manual design of deep learning systems to the design of these automated algorithms. However, once a good algorithm exists, it can be reused for a variety of problems to automatically configure deep learning systems.

Driven by this goal (G1), we explore neural architecture search methods in this thesis. They promise to automate the selection of a network architecture, one of the most important hyperparameters of a deep learning system, at the cost of additional computation. Unfortunately, the amount of additional computation can be so significant that it becomes impractical to apply. This motivates us to strive for NAS algorithms with lower computational requirements (G2).

Finally, a general problem of supervised deep learning is that it requires a large number of labeled samples to achieve good generalization performance. Such a labeled dataset can be very difficult to acquire and motivates approaches that work with smaller datasets as well. We work towards this as our third goal (G3).

Our initial idea to tackle (G1) is to scale neuroevolution up to large DNNs. In a preliminary step, we test if the training of large DNNs with EAs is feasible. Despite the fact that EAs are traditionally applied to problems of much smaller dimensionality, we can successfully train a DNN with almost 100,000 weights using an EA. This is made possible by a GPU-accelerated implementation of the limited evaluation EA

which allows extensive experimentation to properly configure variation operators and hyperparameters. The EA achieves promising DNN training results, but at the same time the process is significantly less efficient than training with SGD. This is mainly due to the fact that the analytical gradient is completely disregarded, while SGD exploits it. In other domains like reinforcement learning where the gradient information is more difficult to exploit, evolutionary approaches compare more favorably to SGD. Both experiments demonstrate that EAs can be a viable alternative to SGD for neural network training even without enormous computational resources, and our open-source implementation opens the door for further research on this topic to find and exploit unique advantages that EAs offer for this problem. Nevertheless, many practical problems that we are interested in work with high-dimensional image data in a supervised learning setting.

Consequently, we change our approach and instead investigate NAS algorithms that use SGD for weight training while optimizing the architecture with evolutionary algorithms. We showcase how a NAS algorithm can create convolutional neural network architectures for image classification (G1) by mutating a genotype that describes a stack of DNN building blocks. The process is expensive, because it requires the training of many candidate network architectures during the evolutionary process. To keep the computational requirements modest, we employ a $(1 + 1)$ EA and extend it with a niching approach to counteract its tendency to converge to local optima. However, the introduction of a mutation with weight inheritance results in a far more significant reduction of this NAS algorithm's computational requirements (G2) while keeping or increasing the network's accuracy. We demonstrate strong data-efficiency gains with this approach on the common CIFAR-10 and CIFAR-100 benchmark datasets and demonstrate that NAS for large-scale networks is possible even in resource-constrained settings.

This is already a very positive result, but further reductions in computational requirements are possible. We investigate a one-shot NAS algorithm, which more closely integrates the weight and architecture optimization, in order to avoid having to train thousands of networks during the search process. We provide a new perspective on configuring a deep MTL system by formulating the process as a multi-task architecture search in which weight sharing between task-specific networks is optimized. By expressing the weight and architecture optimization in a stochastic way, the architecture becomes amenable to optimization by a hybrid NES and SGD algorithm. The architecture space is searched by NES (G1), but we can still exploit the differentiable nature of neural networks (G2) and approximate a weight gradient through Monte-Carlo methods.

By applying our one-shot NAS algorithm to an MTL setting, it becomes possible to work with smaller datasets than usual (G3) by leveraging different related datasets. We demonstrate the applicability of our approach to MTL on three different MTL datasets and significantly outperform similar approaches from literature. It begs

further investigation, to find out what kinds of datasets benefit from MTL and also to allow more heterogeneous task-specific architectures by using a more flexible MTL architecture search space.

Thanks to its modest computational requirements, one-shot NAS is a promising avenue for further research. In particular, we only focus on a single hyperparameter, the network architecture, so far. It would be immensely useful to extend the ideas found in one-shot NAS to other hyperparameters and eliminate the need for manual decisions as much as possible. Orthogonal to that, the hybrid NES and SGD algorithm needs further analysis to identify failure modes and improve upon it. Such an analysis can be performed on problems different from DNN training, but there are also aspects specific to DNNs that require further investigation.

For example, it seems plausible that, given unlimited computational resources, the conventional NAS approach of training architectures from scratch will result in better architectures than one-shot NAS. The latter has to overcome interference between shared weights of competing architectures, which might hurt performance. Luo et al. [75] observe low ranking correlation between networks trained by one-shot NAS and trained-from-scratch versions of the same architectures. Such results are likely dependent on the exact one-shot NAS algorithm and should be investigated in order to find better architectures at low cost.

Part IV

Appendix

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Youhei Akimoto, Shinichi Shirakawa, Nozomu Yoshinari, Kento Uchida, Shota Saito, and Kouhei Nishida. Adaptive stochastic natural gradient method for one-shot neural architecture search. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 171–180, 2019.

[3] Sharib Ali and Felix Zhou, editors. *Proceedings of the 2019 Challenge on Endoscopy Artefacts Detection (EAD2019) co-located with the 16th International Symposium on Biomedical Imaging (ISBI)*, volume 2366 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[4] Dirk V. Arnold. *Noisy optimization with evolution strategies*, volume 8 of *Genetic algorithms and evolutionary computation*. Kluwer, 2002.

[5] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. DENSER: deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines*, 20(1):5–35, 2019.

[6] Marco Baioletti, Gabriele Di Bari, Valentina Poggioni, and Mirco Tracolli. Can differential evolution be an efficient engine to optimize neural networks? In Giuseppe Nicosia, Panos M. Pardalos, Giovanni Giuffrida, and Renato Umeton, editors, *Machine Learning, Optimization, and Big Data - Third International Conference, MOD 2017, Volterra, Italy, September 14-17, 2017, Revised Selected Papers*, volume 10710 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2017.

[7] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[8] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, jul 2019.

[9] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.*, 47:253–279, 2013.

[10] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[11] Hans-Georg Beyer. Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. *Computer methods in applied mechanics and engineering*, 186(2-4):239–267, 2000.

[12] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2787–2794. AAAI Press, 2018.

[13] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[14] Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, Jul 1997.

[15] Sabina Chiaretti, Gina Zini, and Renato Bassan. Diagnosis and subclassification of acute lymphoblastic leukemia. *Mediterranean journal of hematology and infectious diseases*, 6(1):e2014073–e2014073, November 2014.

[16] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. Back to basics: Benchmarking canonical evolution strategies for playing atari. In *Proceedings of the*

Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, pages 1419–1426, 2018.

[17] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep learning for classical japanese literature. *CoRR*, abs/1812.01718, 2018.

[18] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*, pages 2921–2926, 2017.

[19] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 5032–5043, 2018.

[20] Victor Costa, Nuno Lourenço, and Penousal Machado. Coevolution of generative adversarial networks. In Paul Kaufmann and Pedro A. Castillo, editors, *Applications of Evolutionary Computation - 22nd International Conference, EvoApplications 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24-26, 2019, Proceedings*, volume 11454 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2019.

[21] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3123–3131, 2015.

[22] Swagatam Das, Sankha Subhra Mullick, and Ponnuthurai N. Suganthan. Recent advances in differential evolution - an updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 2016.

[23] Travis Desell. Large scale evolution of convolutional neural networks using volunteer computing. In Peter A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 127–128. ACM, 2017.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[25] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017.

[26] Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. Convolution by evolution: Differentiable pattern producing networks. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 109–116. ACM, 2016.

[27] L. J. Fogel. Autonomous automata. *Industrial Research*, 4:14–19, 1962.

[28] Nicolás García-Pedrajas, Domingo Ortiz-Boyer, and César Hervás-Martínez. An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks*, 19(4):514–528, 2006.

[29] Charles J. Geyer. Stat 5421 lecture notes: Exponential families, part i, April 2016.

[30] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[31] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org, 2011.

[32] Faustino J. Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In Johannes Fürnkranz, Tobias

Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, volume 4212 of *Lecture Notes in Computer Science*, pages 654–662. Springer, 2006.

[33] Faustino John Gomez. *Robust Nonlinear Control through Neuroevolution*. PhD thesis, University of Texas at Austin, 2003.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[35] Anubha Gupta and Ritu Gupta, editors. *ISBI 2019 C-NMC Challenge: Classification in Cancer Cell Imaging*. Springer Singapore, 2019.

[36] Ritu Gupta, Pramit Mallick, Rahul Duggal, Anubha Gupta, and Ojaswa Sharma. Stain color normalization and segmentation of plasma cells in microscopic images as a prelude to development of computer assisted automated disease diagnostic tool in multiple myeloma. *Clinical Lymphoma, Myeloma and Leukemia*, 17(1):e99, 2019/03/18 2017.

[37] Hassan Al Hajj, Mathieu Lamard, Pierre-Henri Conze, Soumali Roychowdhury, Xiaowei Hu, Gabija Maršalkaitė, Odysseas Zisimopoulos, Muneer Ahmad Dedmari, Fenqiang Zhao, Jonas Prellberg, Manish Sahu, Adrian Galdran, Teresa Araújo, Duc My Vo, Chandan Panda, Navdeep Dahiya, Satoshi Kondo, Zhengbing Bian, Arash Vahdat, Jonas Bialopetravičius, Evangello Flouty, Chenhui Qiu, Sabrina Dill, Anirban Mukhopadhyay, Pedro Costa, Guilherme Aresta, Senthil Ramamurthy, Sang-Woong Lee, Aurélio Campilho, Stefan Zachow, Shunren Xia, Sailesh Conjeti, Danail Stoyanov, Jogundas Armaitis, Pheng-Ann Heng, William G. Macready, Béatrice Cochener, and Gwenolé Quellec. CATARACTS: Challenge on automatic tool annotation for cataRACT surgery. *Medical Image Analysis*, 52:24–41, feb 2019.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1026–1034, 2015.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778, 2016.

[40] J. Holland. Nonlinear environments permitting efficient adaptation. In *Computer and Information Sciences II*. Academic Press, 1967.

[41] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[42] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[43] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 7132–7141. IEEE Computer Society, 2018.

[44] Abid Hussain, Yousaf Shad Muhammad, M. Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry, and Showkat Gani. Genetic algorithm for traveling salesman problem with modified cycle crossover operator. *Computational Intelligence and Neuroscience*, 2017:1–7, 2017.

[45] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* IEEE, 2003.

[46] OEIS Foundation Inc. The on-line encyclopedia of integer sequences: A003024, 2020. https://oeis.org/A003024.

[47] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[48] Jeremy Irvin, Pranav Rajpurkar, Michael Ko, Yifan Yu, Silviana Ciurea-Ilcus, Chris Chute, Henrik Marklund, Behzad Haghgoo, Robyn L. Ball, Katie S. Shpanskaya, Jayne Seekins, David A. Mong, Safwan S. Halabi, Jesse K. Sandberg, Ricky Jones, David B. Larson, Curtis P. Langlotz, Bhavik N. Patel, Matthew P. Lungren, and Andrew Y. Ng. Chexpert: A large chest radiograph dataset with uncertainty labels and expert comparison. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 590–597. AAAI Press, 2019.

[49] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen

Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017.

[50] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer New York, 2013.

[51] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 1946–1956. ACM, 2019.

[52] Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[53] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017.

[54] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. Neural architecture search with bayesian optimisation and optimal transport. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 2020–2029, 2018.

[55] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 4401–4410. Computer Vision Foundation / IEEE, 2019.

[56] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. *CoRR*, abs/1912.04958, 2019.

[57] A. Emre Kavur, Naciye Sinem Gezer, Mustafa Baris, Pierre-Henri Conze, Vladimir Groza, Duc Duy Pham, Soumick Chatterjee, Philipp Ernst, Savas Özkan, Bora Baydar, Dmitry Lachinov, Shuo Han, Josef Pauli, Fabian Isensee, Matthias Perkonigg, Rachana Sathish, Ronnie Rajan, Sinem Aslan, Debdoot Sheet, Gurbandurdy Dovletov, Oliver Speck, Andreas Nürnberger, Klaus H. Maier-Hein, Gözde B. Akar, Gözde Ünal, Oguz Dicle, and M. Alper Selver. CHAOS challenge - combined (CT-MR) healthy abdominal organ segmentation. *CoRR*, abs/2001.06535, 2020.

[58] Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 1196–1208, 2018.

[59] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[60] Oliver Kramer. Evolution of convolutional highway networks. In Kevin Sim and Paul Kaufmann, editors, *Applications of Evolutionary Computation - 21st International Conference, EvoApplications 2018, Parma, Italy, April 4-6, 2018, Proceedings*, volume 10784 of *Lecture Notes in Computer Science*, pages 395–404. Springer, 2018.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

[62] Kim W. C. Ku, Man-Wai Mak, and Wan-Chi Siu. A study of the lamarckian evolution of recurrent neural networks. *IEEE Trans. Evolutionary Computation*, 4(1):31–42, 2000.

[63] R. D. Labati, V. Piuri, and F. Scotti. All-idb: The acute lymphoblastic leukemia image database for image processing. In *2011 18th IEEE International Conference on Image Processing*, pages 2045–2048, Sep. 2011.

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

[65] Sang-Woo Lee, Chung-yeon Lee, Dong-Hyun Kwak, Jiwon Kim, Jeonghee Kim, and Byoung-Tak Zhang. Dual-memory deep learning architectures for lifelong learning of everyday human behaviors. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1669–1675, 2016.

[66] Joel Lehman, Jay Chen, Jeff Clune, and Kenneth O. Stanley. ES is more than just a traditional finite-difference approximator. In Hernán E. Aguirre and Keiki Takadama, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 450–457. ACM, 2018.

[67] Karel Lenc, Erich Elsen, Tom Schaul, and Karen Simonyan. Non-differentiable supervised learning with evolution strategies and hybrid methods. *CoRR*, abs/1906.03139, 2019.

[68] Hongzhou Lin and Stefanie Jegelka. Resnet with one-neuron hidden layers is a universal approximator. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 6172–6181, 2018.

[69] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part I*, volume 11205 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2018.

[70] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[71] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[72] Xiaodong Liu, Jianfeng Gao, Xiaodong He, Li Deng, Kevin Duh, and Ye-Yi Wang. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 912–921, 2015.

[73] Yongxi Lu, Abhishek Kumar, Shuangfei Zhai, Yu Cheng, Tara Javidi, and Rogério Schmidt Feris. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 1131–1140, 2017.

[74] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 6231–6239, 2017.

[75] Renqian Luo, Tao Qin, and Enhong Chen. Understanding and improving one-shot neural architecture optimization. *CoRR*, abs/1909.10815, 2019.

[76] H. T. Madhloom, S. A. Kareem, and H. Ariffin. A robust feature extraction and selection method for the recognition of lymphocytes versus acute lymphoblastic leukemia. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 330–335, Nov 2012.

[77] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943.

[78] Jieru Mei, Yingwei Li, Xiaochen Lian, Xiaojie Jin, Linjie Yang, Alan L. Yuille, and Jianchao Yang. Atomnas: Fine-grained end-to-end neural architecture search. In *8th International Conference on Learning Representations, ICLR 2020*, 2020.

[79] Elliot Meyerson and Risto Miikkulainen. Beyond shared hierarchies: Deep multitask learning through soft layer ordering. In *6th International Conference*

on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, 2018.

[80] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[81] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 3994–4003, 2016.

[82] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[83] Subrajeet Mohapatra, Dipti Patra, and Sanghamitra Satpathy. An ensemble classifier system for early diagnosis of acute lymphoblastic leukemia in blood microscopic images. *Neural Computing and Applications*, 24(7):1887–1904, Jun 2014.

[84] David E. Moriarty and Risto Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1-3):11–32, 1996.

[85] Gregory Morse and Kenneth O. Stanley. Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 477–484. ACM, 2016.

[86] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. In *8th International Conference on Learning Representations, ICLR 2020*, 2020.

[87] Yann Ollivier, Ludovic Arnold, Anne Auger, and Nikolaus Hansen. Information-geometric optimization algorithms: A unifying picture via invariance principles. *J. Mach. Learn. Res.*, 18:18:1–18:65, 2017.

[88] Matt Parker and Bobby D. Bryant. Lamarckian neuroevolution for visual control in the quake II environment. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2009, Trondheim, Norway, 18-21 May, 2009*, pages 2630–2637. IEEE, 2009.

[89] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[90] Caroline Petitjean, Su Ruan, Zoé Lambert, and Bernard Dubray, editors. *Proceedings of the 2019 Challenge on Segmentation of THoracic Organs at Risk in CT Images, SegTHOR@ISBI 2019, April 8, 2019*, volume 2349 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[91] Kilian M. Pohl, Wesley K. Thompson, Ehsan Adeli, and Marius George Linguraru, editors. *Adolescent Brain Cognitive Development Neurocognitive Prediction - First Challenge, ABCD-NP 2019, Held in Conjunction with MICCAI 2019, Shenzhen, China, October 13, 2019, Proceedings*, volume 11791 of *Lecture Notes in Computer Science*. Springer, 2019.

[92] Jonas Prellberg and Oliver Kramer. Lamarckian evolution of convolutional neural networks. In Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and Darrell Whitley, editors, *Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part II*, volume 11102 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 2018.

[93] Jonas Prellberg and Oliver Kramer. Limited evaluation evolutionary optimization of large neural networks. In Frank Trollmann and Anni-Yasmin Turhan, editors, *KI 2018: Advances in Artificial Intelligence - 41st German Conference on AI, Berlin, Germany, September 24-28, 2018, Proceedings*, volume 11117 of *Lecture Notes in Computer Science*, pages 270–283. Springer, 2018.

[94] Jonas Prellberg and Oliver Kramer. Acute lymphoblastic leukemia classification from microscopic images using convolutional neural networks. In Anubha Gupta and Ritu Gupta, editors, *ISBI 2019 C-NMC Challenge: Classification in Cancer Cell Imaging*, pages 53–61, Singapore, 2019. Springer Singapore.

[95] Jonas Prellberg and Oliver Kramer. Learned weight sharing for deep multi-task learning by natural evolution strategy and stochastic gradient descent. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, UK, July 19-24, 2020*. IEEE, 2020.

[96] Ching-Hon Pui. *Acute Lymphoblastic Leukemia*, pages 39–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[97] Lorenzo Putzu and Cecilia Di Ruberto. White blood cells identication and classication from leukemic blood image. In *International Work-Conference on Bioinformatics and Biomedical Engineering, IWBBIO 2013, Granada, Spain, March 18-20, 2013. Proceedings*, pages 99–106, 2013.

[98] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911. PMLR, 2017.

[99] I. Rechenberg, B.F. Toms, and Royal Aircraft Establishment. *Cybernetic Solution Path of an Experimental Problem:*. Library translation / Royal Aircraft Establishment. Ministry of Aviation, 1965.

[100] Amjad Rehman, Naveed Abbas, Tanzila Saba, Syed Ijaz ur Rahman, Zahid Mehmood, and Hoshang Kolivand. Classification of acute lymphoblastic leukemia using deep learning. *Microscopy Research and Technique*, 81(11):1310–1317, 2018.

[101] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning*. The MIT Press, 2018.

[102] Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. Routing networks: Adaptive selection of non-linear functions for multi-task learning. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[103] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[104] Sebastian Ruder, Joachim Bingel, Isabelle Augenstein, and Anders Søgaard. Sluice networks: Learning what to share between loosely related tasks. *CoRR*, abs/1705.08142, 2017.

[105] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, oct 1986.

[106] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein,

Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[107] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017.

[108] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 2488–2498, 2018.

[109] N. Saravanan and David B. Fogel. Evolving neural control systems. *IEEE Expert*, 10(3):23–27, 1995.

[110] Takahiro Sasaki and Mario Tokoro. Comparison between lamarckian and darwinian evolution on a model using neural networks and genetic algorithms. *Knowl. Inf. Syst.*, 2(2):201–222, 2000.

[111] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[112] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: mit einer vergleichenden Einführung in die Hill-Climbing-und Zufallsstrategie*, volume 1. Springer, 1977.

[113] Sarmad Shafique and Samabia Tehsin. Acute lymphoblastic leukemia detection and classification of its subtypes using pretrained deep convolutional neural networks. *Technology in Cancer Research & Treatment*, 17:1533033818802789, 2018. PMID: 30261827.

[114] Shinichi Shirakawa, Yasushi Iwata, and Youhei Akimoto. Dynamic optimization of neural network structures using probabilistic modeling. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 4074–4082. AAAI Press, 2018.

[115] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran,

Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, dec 2018.

[116] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[117] David R. So, Quoc V. Le, and Chen Liang. The evolved transformer. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 5877–5886. PMLR, 2019.

[118] S Spigler, M Geiger, S d'Ascoli, L Sagun, G Biroli, and M Wyart. A jamming transition from under- to over-parametrization affects generalization in deep learning. *Journal of Physics A: Mathematical and Theoretical*, 52(47):474001, oct 2019.

[119] Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.

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

[121] 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.

[122] Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant G. Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*, pages 569–577. Morgan Kaufmann, 2002.

[123] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, jun 2002.

[124] Kenneth O. Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *J. Artif. Intell. Res.*, 21:63–100, 2004.

[125] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567, 2017.

[126] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In Peter A. N. Bosman, editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 497–504. ACM, 2017.

[127] Yi Sun, Daan Wierstra, Tom Schaul, and Jürgen Schmidhuber. Stochastic search using the natural gradient. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 1161–1168, 2009.

[128] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015.

[129] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.

[130] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J. Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 1553–1561, 2017.

[131] Dirk Thierens. Non-redundant genetic coding of neural networks. In Toshio Fukuda and Takeshi Furuhashi, editors, *Proceedings of 1996 IEEE International Conference on Evolutionary Computation, Nayoya University, Japan, May 20-22, 1996*, pages 571–575. IEEE, 1996.

[132] Jamal Toutouh, Erik Hemberg, and Una-May O'Reilly. Spatial evolutionary generative adversarial networks. In Anne Auger and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 472–480. ACM, 2019.

[133] Vladimír Ulman, Martin Maška, Klas E G Magnusson, Olaf Ronneberger, Carsten Haubold, Nathalie Harder, Pavel Matula, Petr Matula, David Svoboda,

Miroslav Radojevic, Ihor Smal, Karl Rohr, Joakim Jaldén, Helen M Blau, Oleh Dzyubachyk, Boudewijn Lelieveldt, Pengdong Xiao, Yuexiang Li, Siu-Yeung Cho, Alexandre C Dufour, Jean-Christophe Olivo-Marin, Constantino C Reyes-Aldasoro, Jose A Solis-Lemus, Robert Bensch, Thomas Brox, Johannes Stegmaier, Ralf Mikut, Steffen Wolf, Fred A Hamprecht, Tiago Esteves, Pedro Quelhas, Ömer Demirel, Lars MalmstrÃ¶m, Florian Jug, Pavel Tomancak, Erik Meijering, Arrate Muñoz-Barrutia, Michal Kozubek, and Carlos Ortiz de Solorzano. An objective comparison of cell-tracking algorithms. *Nature Methods*, 14(12):1141–1152, oct 2017.

[134] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[135] Phillip Verbancsics and Josh Harguess. Image classification using generative neuro evolution for deep learning. In *2015 IEEE Winter Conference on Applications of Computer Vision, WACV 2015, Waikoloa, HI, USA, January 5-9, 2015*, pages 488–493. IEEE Computer Society, 2015.

[136] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, MichaÃ«l Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario WÃŒnsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, oct 2019.

[137] L. H. S. Vogado, R. D. M. S. Veras, A. R. Andrade, F. H. D. D. Araujo, R. R. V. e. Silva, and K. R. T. Aires. Diagnosing leukemia in blood smear images using an ensemble of classifiers and pre-trained convolutional neural networks. In *2017 30th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 367–373, Oct 2017.

[138] Chaoyue Wang, Chang Xu, Xin Yao, and Dacheng Tao. Evolutionary generative adversarial networks. *IEEE Trans. Evolutionary Computation*, 23(6):921–934, 2019.

[139] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1995–2003. JMLR.org, 2016.

[140] Shimon Whiteson and Peter Stone. Evolutionary function approximation for reinforcement learning. *J. Mach. Learn. Res.*, 7:877–917, 2006.

[141] A. P. Wieland. Evolving neural network controllers for unstable systems. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*. IEEE, 1991.

[142] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *J. Mach. Learn. Res.*, 15(1):949–980, 2014.

[143] Jiqing Wu, Zhiwu Huang, Dinesh Acharya, Wen Li, Janine Thoma, Danda Pani Paudel, and Luc Van Gool. Sliced wasserstein generative models. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 3713–3722. Computer Vision Foundation / IEEE, 2019.

[144] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995, 2017.

[145] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[146] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[147] Anil Yaman, Decebal Constantin Mocanu, Giovanni Iacca, George H. L. Fletcher, and Mykola Pechenizkiy. Limited evaluation cooperative co-evolutionary differential evolution for large-scale neuroevolution. In Hernán E. Aguirre and Keiki Takadama, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 569–576. ACM, 2018.

[148] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information*

Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada, pages 3320–3328, 2014.

[149] Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. In *8th International Conference on Learning Representations, ICLR 2020*, 2020.

[150] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833. Springer, 2014.

[151] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[152] Xingwen Zhang, Jeff Clune, and Kenneth O. Stanley. On the relationship between the openai evolution strategy and stochastic gradient descent. *CoRR*, abs/1712.06564, 2017.

[153] Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Facial landmark detection by deep multi-task learning. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VI*, pages 94–108, 2014.

[154] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[155] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8697–8710, 2018.