A Survey on Neural Architecture Search

Martin Wistuba MARTIN.WISTUBA@IBM.COM
IBM Research - Ireland
Dublin Technology Campus, Damastown Ind. Park
Mulhuddart, Dublin 15, Ireland

Ambrish Rawat AMBRISH.RAWAT@IE.IBM.COM
IBM Research - Ireland
Dublin Technology Campus, Damastown Ind. Park
Mulhuddart, Dublin 15, Ireland

Tejaswini Pedapati TEJASWINIP@US.IBM.COM
IBM Research AI
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA

Abstract

The growing interest in both the automation of machine learning and deep learning has inevitably led to the development of automated methods for neural architecture optimization. The choice of the network architecture has proven to be critical, and many advances in deep learning spring from its immediate improvements. However, deep learning techniques are computationally intensive and their application requires a high level of domain knowledge. Therefore, even partial automation of this process would help make deep learning more accessible to both researchers and practitioners. With this survey, we provide a formalism which unifies and categorizes the landscape of existing methods along with a detailed analysis that compares and contrasts the different approaches. We achieve this via a discussion of common architecture search spaces and architecture optimization algorithms based on principles of reinforcement learning and evolutionary algorithms along with approaches that incorporate surrogate and one-shot models. Additionally, we address the new research directions which include constrained and multi-objective architecture search as well as automated data augmentation, optimizer and activation function search.

Keywords: Neural Architecture Search, Automation of Machine Learning, Deep Learning, Reinforcement Learning, Evolutionary Algorithms, Constrained Optimization, Multi-Objective Optimization

1. Introduction

Deep learning methods are very successful in solving tasks in machine translation, image and speech recognition or reinforcement learning in general. This success is often attributed to their ability of automatically extracting features from unstructured data such as audio, image and text. We are currently witnessing this paradigm shift from the laborious job of manual feature engineering for unstructured data to engineering network components and architectures for deep learning methods. While architecture modifications do result in significant gains in the performance of deep learning methods, the search for suitable architectures is in itself a time-consuming, arduous and error-prone task. Within the last two years there has been an insurgence in research efforts by the machine learning community that seeks to automate this search process. Arguably, the work by Zoph and Le (2017) marks the beginning of these efforts where their work demonstrated that good architectures can be discovered with the use of reinforcement learning algorithms. Shortly thereafter, Real et al. (2017) showed that similar results could also be achieved by the hitherto well studied approaches in neuroevolution (Floreano et al., 2008). However, both these search methods require thousands of GPU hours. Hence, many of the subsequent works attempt to reduce this computational burden. Along these lines, most successful algorithms leverage from the principle of reusing the learned model parameters, the most notable mentions being the works of Cai et al. (2018a) and Pham et al. (2018). Cai et al. (2018a) propose to begin the search with a simple architecture and progressively increase its breadth and depth with function-preserving operations. The popular and faster search method by Pham et al. (2018) achieves this by formulating an over-parameterized architecture that encompasses all architectures spanned by the search space. At every time step of their algorithm, a smaller part of this large architecture is sampled and trained. Training is done such that the weights are shared across sampled architectures which reduces the search effort to about the same as training a single architecture.

The design of the search space forms another key component of neural architecture search. In addition to speeding up the search process, this influences the duration of the search and the quality of the solution. In the earlier works on neural architecture search, the spaces were designed to primarily search for sequential architectures. However, with branched handcrafted architectures surpassing the classical networks in terms of performance, appropriate search spaces were proposed shortly after the initial publications and these have since become a norm in this field (Zoph et al., 2018).

In parallel to these developments, researchers have broadened the horizons of neural architecture search to incorporate objectives that go beyond reducing the search time and generalization error of the found architectures. Methods that simultaneously handle multiple objective functions have become relevant. Notable works include methods that attempt to limit the number of model parameters or the like, for efficient deployment on mobile devices (Tan et al., 2018; Kim et al., 2017). Furthermore, the developed techniques for architecture search have been extended for advanced automation of other related components of deep learning. For instance, the search for activation functions (Ramachandran et al., 2018) or suitable data augmentation (Cubuk et al., 2018a).

Currently, the automation of deep learning in the form of neural architecture search is one of the fastest developing areas of machine learning. With new papers emerging on `arXiv.org` each week and major conferences publishing a handful of interesting work, it is easy to lose track. With this survey, we provide a formalism which unifies the landscape of existing methods. This formalism allows us to critically examine the different approaches and understand the benefits of different components that contribute to the design and success of neural architecture search. Along the way, we also seek to dispel some popular misconceptions and highlight some pitfalls in the current trends of architecture search. We supplement our criticism with suitable experiments.

Our review is divided into four sections. In Section 2, we discuss various architecture search spaces that have been proposed over time. We use Section 3 to formally define the problem of architecture search. Then we identify four typical types of optimization methods: reinforcement learning, evolutionary algorithms, surrogate model-based optimization, and one-shot architecture search. We define these optimization procedures and associate them to existing work and discuss it. Section 4 highlights the architecture search, considering constraints and multiple objective functions. Finally, the influence of search procedures on related areas is discussed in Section 5.

2. Neural Architecture Search Space

From a computational standpoint, neural networks represent a function that transforms input variables $\boldsymbol{x}$ to output variables $\hat{\boldsymbol{y}}$ through a series of operations. This can be formalized in the language of computational graphs (Bauer, 1974) where neural networks are represented as directed acyclic graphs with a set of nodes Z. Each node $\boldsymbol{z}^{(k)}$ represents a tensor and is associated with an operation $o^{(k)} \in O$ on its set of parent nodes $I^{(k)}$ (Goodfellow et al., 2016). The only exception is the input node $\boldsymbol{x}$ which has neither a set of parent nodes nor an operation associated to it and is only considered as an input to other nodes. The computation at a node k amounts to

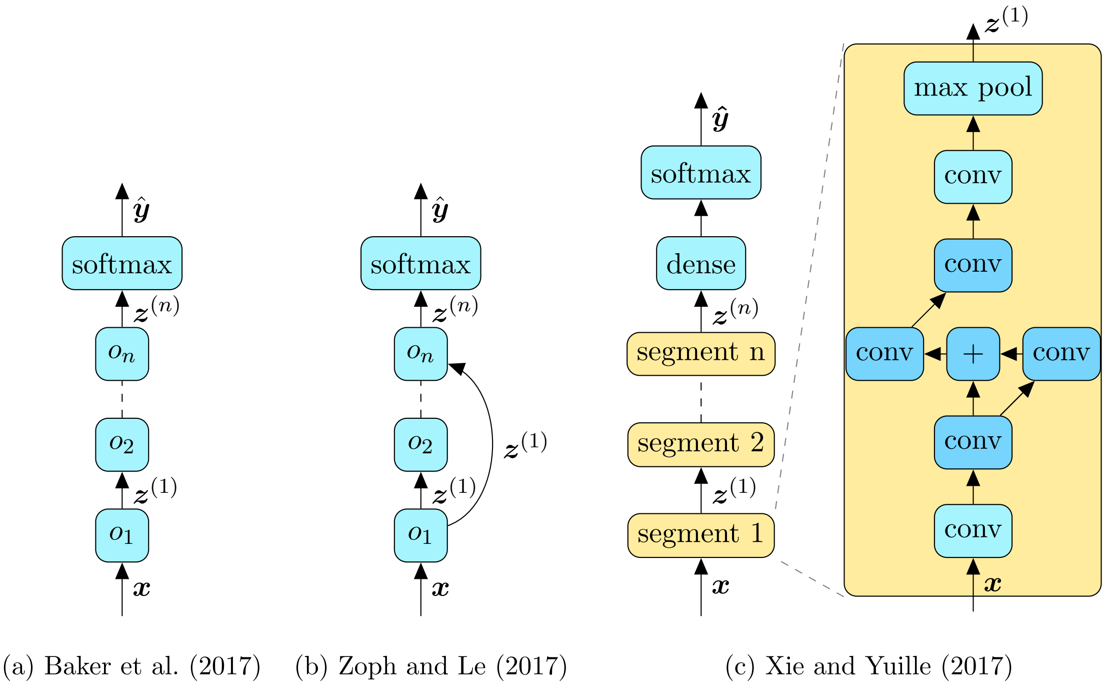
$$\boldsymbol{z}^{(k)} = o^{(k)}(I^{(k)}) \,. \tag{1}$$

Examples for operations are unary operations such as convolutions, pooling, activation functions or multivariate operations such as concatenation or addition. For notational convenience, we fix the unary operations to use concatenation as a merge operation when acting on a set of multiple inputs and often omit it in the representation. Note that for a neural network, any representation that comprises of a specification of the parents and the operation for each node completely defines its architecture. We refer to such a representation as $\boldsymbol{\alpha}$ and use this to unify and outline the different neural architecture search methods under the same framework.

A *neural architecture search space* is a subspace of this general definition of neural architectures. Its space of operations can be limited and certain constraints may be imposed on the architectures. In the rest of this survey, we use search space to refer to the set of feasible solutions of a neural architecture search method. Most work on neural architecture search addresses the automated search of architectures for image recognition problems and therefore convolutional neural networks (CNNs) (LeCun et al., 1998). The corresponding search spaces can be broadly classified into two categories. The first category of search spaces is defined for the graphs that represent an entire neural architecture which we refer to as the global search space and discuss in Section 2.1. We discuss the second category in Section 2.2 which assumes that an architecture is a combination of few cells which are repeated to build the complete network. An objective comparison of these search spaces is provided in Section 2.3. Another common problem tackled with neural architecture search is language modeling by means of recurrent neural networks (RNNs). We discuss the most common search space in Section 2.4.

2.1 Global Search Space

Instances belonging to the global search space admit large degrees of freedom regarding the arrangement of operations. An *architecture template* may be assumed which limits the freedom of admissible structural choices within an architecture definition. This template



(a) Baker et al. (2017) (b) Zoph and Le (2017) (c) Xie and Yuille (2017)

Figure 1: Global search spaces: (a) sequential search space, (b) same with skips, (c) architecture template, only the connections between the dark blue operations are not fixed.

often fixes certain aspects of the network graph. For instance, it may divide the architecture graph into several segments or enforce specific constraints on operations and connections both within and across these segments, thereby limiting the type of architectures that belong to a search space. Figure 1 illustrates examples of architectures from such template-constrained search spaces. Here, we exclude constraints which enforce a predetermined repeating pattern of subgraphs and dedicate a separate section for the discussion of such search spaces in Section 2.2.

The simplest example of a global search space is the sequential search space as shown in Figure 1a. In its most primitive form, this search space consists of architectures that can be represented by an arbitrary sequence of ordered nodes where $\boldsymbol{z}^{(k-1)}$ is the only parent of the node $\boldsymbol{z}^{(k)}$. The data flow for neural networks with such architectures simplifies to

$$\boldsymbol{z}^{(k)} = o^{(k)}\left(\left\{\boldsymbol{z}^{(k-1)}\right\}\right) . \tag{2}$$

Baker et al. (2017) explore this search space. They define a set of operations which includes convolutions, pooling, linear transformations with activation (dense layers), and global average pooling with different hyperparameter settings such as number of filters, kernel size, stride and pooling size. Furthermore, they consider additional constraints so as to exclude certain architectures that correspond to patently poor or computationally expensive neural network models. For instance, architectures with pooling as the first operation do not belong to their search space. Furthermore, architectures with dense (fully connected) layers applied to high-resolution feature maps, or as feature transformations before other operations like convolutions are excluded from their search space.

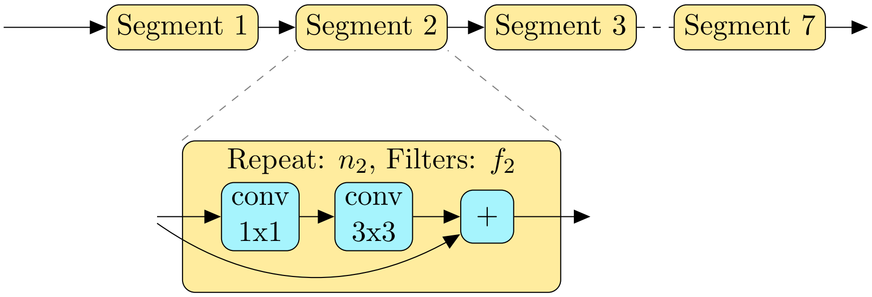


Figure 2: Tan et al. (2018) propose to factorize the architecture in different segments. Each segment i has its own pattern (blue operations) which is repeated n_i times and has f_i filters.

In a parallel work around the same time, Zoph and Le (2017) define a relaxed version of the sequential search space. By permitting arbitrary skip connections to exist between the ordered nodes of a sequential architecture, members belonging to this search space exhibit a wider variety of designs. However, the operation set is restricted to use only convolutions with different hyperparameter settings. For an architecture in this search space, the set of parents of a node $\boldsymbol{z}^{(k)}$ necessarily contains $\boldsymbol{z}^{(k-1)}$ with possible inclusion of other ancestor nodes,

$$\boldsymbol{z}^{(k)} = o^{(k)}\left(\left\{\boldsymbol{z}^{(k-1)}\right\} \cup \left\{\boldsymbol{z}^{(i)} \mid \alpha_{i,k} = 1,\ i < k-1\right\}\right) . \tag{3}$$

For these nodes of the latter type, the merge operation is fixed as concatenation. Figure 1b provides an example of such a skip connection. As discussed earlier, we omit the explicit representation of the concatenation.

Xie and Yuille (2017) consider a similar search space which uses summation instead of concatenation as the merging operation. Furthermore, the architectures in this search space are no longer sequential in the sense that the set of parents for a node $\boldsymbol{z}^{(k)}$ does not necessarily contain the node $\boldsymbol{z}^{(k-1)}$. Moreover this search space incorporates a template which separates architectures into sequentially connected segments (e.g. three segments are used for image recognition on the CIFAR-10 dataset (Krizhevsky, 2009)). Each segment is parameterized by a set of nodes with convolutions as their operation. As part of the template the segments begin with a convolution and conclude with a max pooling operation with a stride of two to reduce feature dimensions (see Figure 1c).

Additionally, the maximum number of convolution operations along with their number of filters for each segment is also fixed as part of the template (for the case of CIFAR-10 this is fixed to three convolutions of filter size 64 for the first, four convolutions of filter size 128 for the second, and five convolutions of filter size 256 for the last segment). With the operations fixed, an adjacency matrix defines the connections in the directed acyclic graphs corresponding to each segment.

An alternative work by Tan et al. (2018) is motivated to look for neural network models for mobile devices that perform efficiently on multiple fronts which include accuracy, inference time and number of parameters. They design a suitable search space for this purpose that consists of architectures with a hierarchical representation (Figure 2). Architectures in this search space are also formed by sequentially connecting segments. Segments are structured to have repeating patterns of operations and each segment is parameterized by

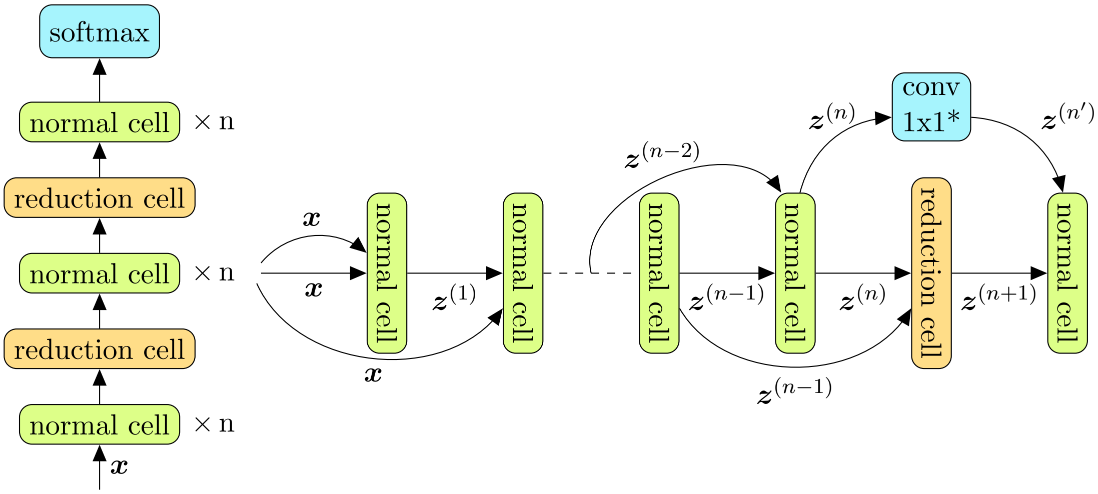


Figure 3: Structure of the NASNet search space instances. n normal cells followed by a reduction cell. This sequence is repeated several times, the reduction cell might be repeated. This decision is a hyperparameter and depends on the image resolution. The 1x1* convolution is a special operation which converts $\boldsymbol{z}^{(n)}$ to match the shape of $\boldsymbol{z}^{(n+1)}$.

the choice of operations and the number of repetitions of the patterns. The structure of patterns itself is simplified to be sequentially connected operations with an optional skip connection and an optional choice of stride. Although they simplify the per-segment search space, they allow segments to be different which they argue provides them the necessary flexibility to tackle multi-objective designs.

2.2 Cell-Based Search Space

A cell-based search space builds upon the observation that many effective handcrafted architectures are based on repeating a fixed structure. Such architectures often consist of smaller-sized graphs that are stacked to form the larger architecture. Across the literature these repeating structures have been interchangeably referred to as cells, blocks or units, we refer to them as cells. Not only is this design known to yield high performance, it also enables easy generalization to other datasets and tasks as these units can be flexibly stacked to build larger or smaller networks. Zoph et al. (2018) was one of the first works to explore such a search space leading to the popular architecture called NASNet. Post the development of this work, other cell-based search spaces have been proposed. Many later approaches that utilize a cell-based search space broadly stick to the structure proposed by Zoph et al. (2018) with small modifications to the choice of operations and cell combination strategies. In the remaining section we define and discuss these different approaches to cell-based search spaces.

In the cell-based search space a network is constructed by repeating a structure called a cell in a prespecified arrangement as determined by a template (see Figure 3). A cell is often a small directed acyclic graph. While the topology of the cell is maintained across the network, its hyperparameters are often varied. For instance the number of filters for

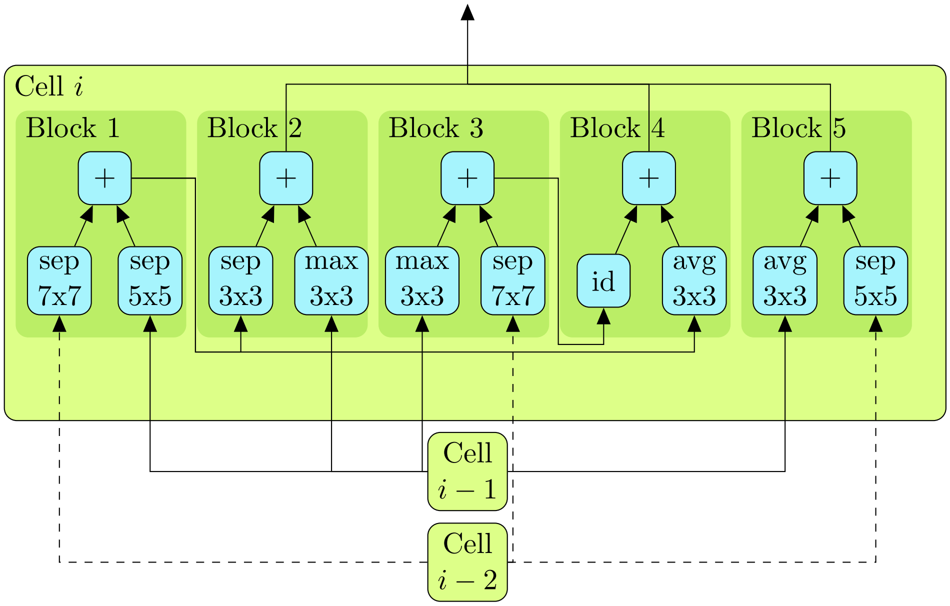


Figure 4: Reduction cell of the NASNet-A architecture (Zoph et al., 2018) as one example how a cell in the NASNet search space can look like. Blocks can be used as input for other blocks (e.g. block 1 and 3), unused blocks are concatenated and are the output of the cell.

the convolution operations can differ across different cells and a template usually specifies the fixed policy for varying these across the network structures. One aspect of approaches to cell-based search space that demands attention is their handling of dimensions and the stacking of different cells. While some approaches have dedicated cells to take care of dimension reduction like the reduction cells of Zoph et al. (2018), others achieve this by introducing pooling operations between different units in the network (Zhong et al., 2018). The prespecified template for connecting the various cells localizes the search to structures within a cell. Often the template also includes a set of initial convolution layers which capture low-level features for the given dataset. It is also worth remarking that most approaches that seek to learn a topology limit the hyperparameter choice to smaller values during the search process thereby making the search process efficient. The final proposed network is trained with a higher number of filters (f) and more repetitions of the cells (n) with an aim to achieve better performance.

Zoph et al. (2018) are one of the first to propose a cell-based approach, the *NASNet search space*. The architectures in this search space consider a template of cells as visualized in Figure 3. This search space distinguishes two cell types, namely normal cells and reduction cells, which handle the feature dimensions across the architecture. Operations in a normal cell have a stride of one and do not change the dimensions of the feature maps, whereas the first operations in a reduction cell have a stride of two and halve the spatial dimensions of the feature maps. Each cell consists of b blocks, where b is a search space hyperparameter and $b = 5$ is the most common choice. Each block is completely defined by its two inputs and the corresponding operations. The computation flow for a block is

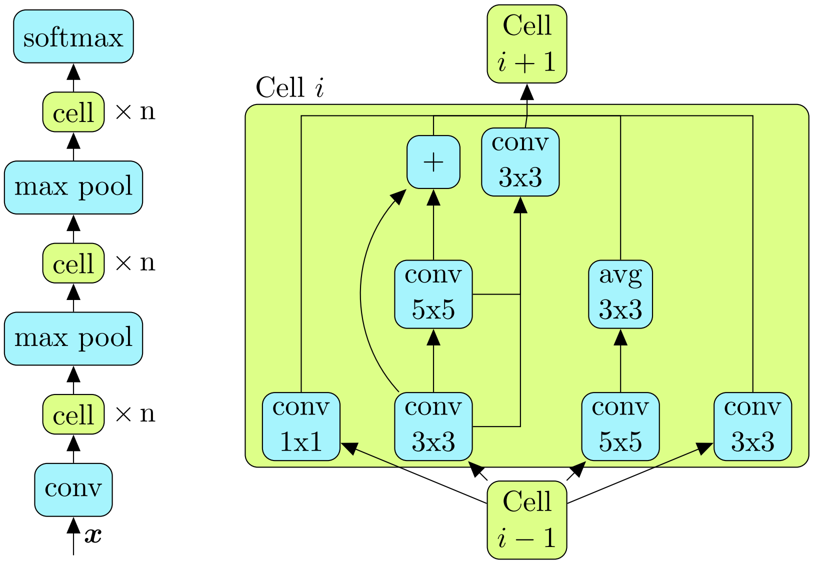


Figure 5: Architecture template (left) and the Block-QNN-B cell discovered by Zhong et al. (2018) (right).

defined by

$$\boldsymbol{z}^{(\text{block}_i)} = o^{(i_1)}\left(\left\{\boldsymbol{z}^{(i_1)}\right\}\right) + o^{(i_2)}\left(\left\{\boldsymbol{z}^{(i_2)}\right\}\right) . \tag{4}$$

An example for the cell structure is visualized in Figure 4. In their original formulation, Zoph et al. (2018) also consider concatenation as a possible merge operation of $o^{(i_1)}$ and $o^{(i_2)}$. However in their experiments they noticed that architectures with concatenation operation are dominated by those with summation in the search process. Therefore, later works such as Liu et al. (2018a) fix this decision to summation. For a block, the set of possible inputs include the output of one of the previous two cells and the output of a previously defined block within a cell. By including the output from previous two cells as candidates for the input to the block, this search space includes instances with skip connections across cells. The output of a cell is determined as concatenation of the outputs of all blocks within the cell which do not serve as inputs to any other block. Cells from this search space differ in their choice of the inputs and the operations for the different blocks.

Zhong et al. (2018) entertain a similar cell-based search space as shown in Figure 5. The key differences lie in their definition of cells and the use of fixed max-pooling layers as opposed to reduction cells for handling feature dimensions in the architecture template. Since they do not decompose the cell structure any further, graphs comprising of arbitrary connections between different nodes are admissible as cells in this search space which makes this cell structure significantly more complex. This alternate definition of cell admits higher degrees of freedom than the normal cell of the NASNet search space, for instance the number of operations within each cell is not fixed. In addition to the number of operations, the instances of cells also differ in the type of operations and their input. This search space formulation does not allow skip connections across different cells. An example architecture discovered is presented in Figure 5.

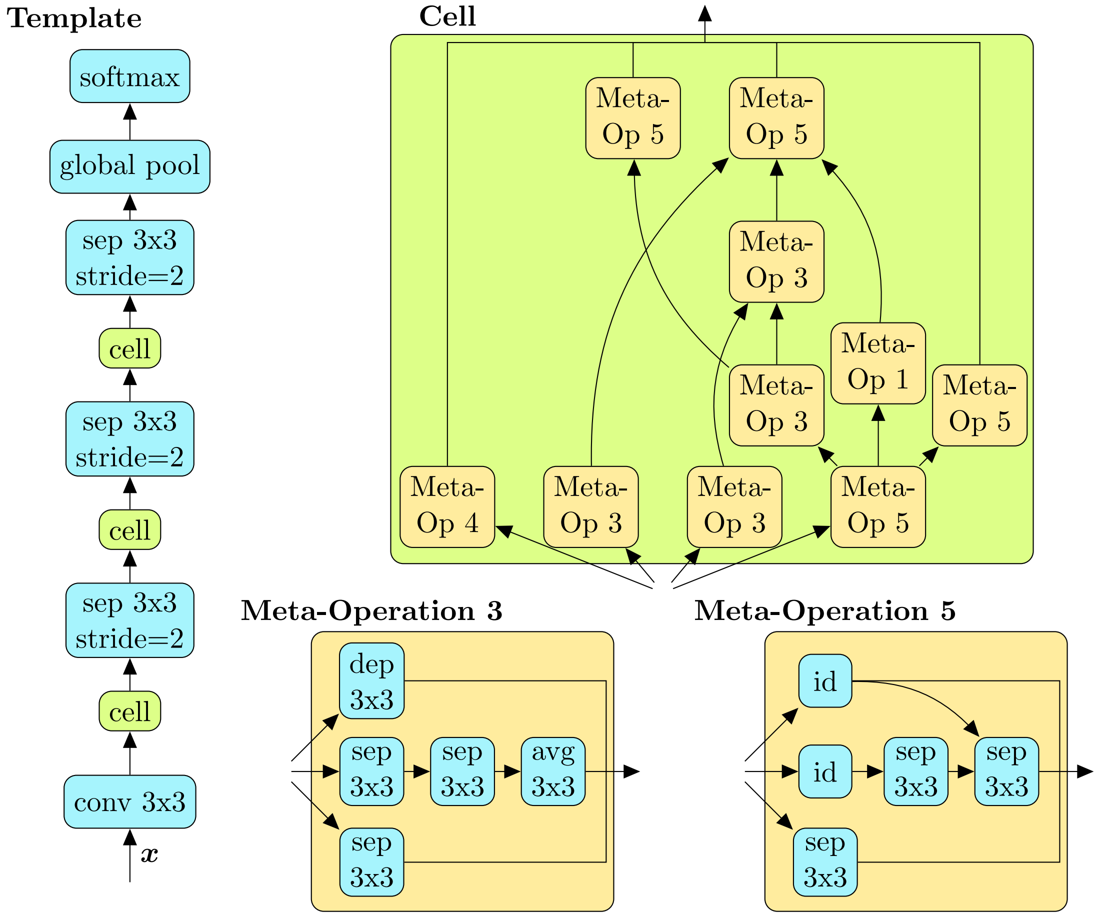


Figure 6: The search space proposed by Liu et al. (2018b) is based on an architecture template (left) which defines the sequence of cells and reduction operations. The choices made about an architecture is the set of meta-operations and their arrangement within the cell (right).

Liu et al. (2018b) propose a search space similar to the one proposed by Zhong et al. (2018). An architecture template describes the high-level definition of an architecture as shown in Figure 6. The main difference is that the search space is decomposed into a hierarchical search space. The first level of hierarchy consists of defining meta-operations. A meta-operation is the connection of few operations to a larger segment, where the number of the meta-operations is limited. The second level represents the cell by connections between the meta-operations.

The cell-based design paradigm has also been used for defining search spaces that are suitable for mobile devices. Dong et al. (2018) propose a search space specifically aimed at meeting such requirements which include objectives like fewer parameters and fast inference time. They are searching for cells without branches and a fixed internal structure that alternates two normalization (e.g. batch normalization) and convolution layers. All operations are densely connected (Huang et al., 2017) as shown in Figure 7 which also applies for those

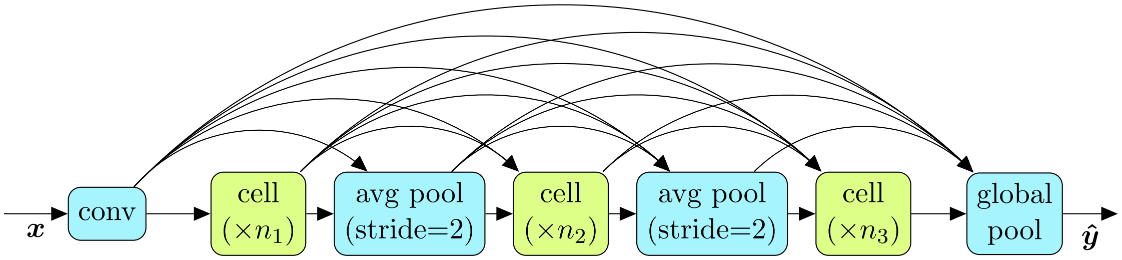


Figure 7: Mobile search spaces as used by Dong et al. (2018). The entire network including the cells are densely connected.

in the cell. The number of choices is significantly smaller than the previously discussed cell-based search spaces, making this a less challenging optimization task.

2.3 Global vs. Cell-Based Search Space

So far in the vast literature of neural architecture search there has been no detailed comparative study of the different search space. However, cell-based search spaces, in particular the NASNet search space, are the most popular choices for developing new methods. Most works that examine both search spaces support this choice by providing empirical evidence that better results can be obtained in the cell-based search space (Pham et al., 2018). Regardless, cell-based search spaces benefit from the advantage that discovered architectures can be easily transferred across datasets. Moreover, the complexity of the architectures can be varied almost arbitrarily by changing the number of filters and cells. In general, the architectures belonging to the global search space do not exhibit all these properties, but specific cases may benefit from some of them. For instance, architectures can be naturally modified by varying the number of filters, but transferring a discovered architecture to a new datasets with different input shapes or deepening the architecture is non trivial. Interestingly, Tan et al. (2018) endorse the use of global search space when searching for mobile architectures. They base this argument on a hypothesis that diversity of layers is critical for achieving both high accuracy and low latency for deployment in mobile devices and that these cannot be provided by cell-based search spaces. Hu et al. (2018) remark on the importance of selecting initial architectures for searching in the global search space and show that architectures with comparable performance to the ones discovered in cell-based search spaces can be easily arrived at with appropriate initial conditions. This is an interesting insight which might revive the global search space in the development of new methods. However, the choice of useful initial architecture might be task-dependent and the guidelines for its selection remain unclear.

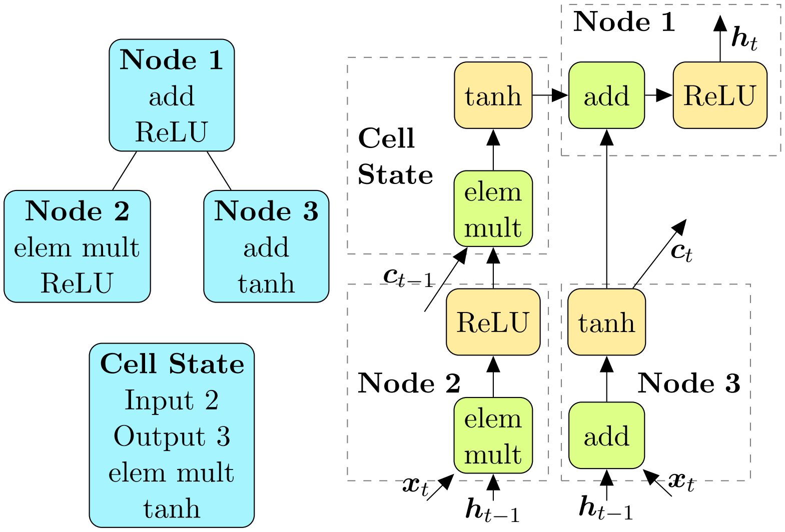


Figure 8: On the left is an example for a full binary tree describing a recurrent cell. Each node has a merging and an activation function associated. The cell state information is stored independently. It points to two nodes from the binary tree which are associated to the input and output as well as a merging and activation function. The right shows how this representation is translated to a computational graph.

2.4 Search Space for Recurrent Cells

Another key contribution from Zoph and Le (2017) work was the definition of a search space for recurrent cells. A recurrent cell (g) can be described by

$$h_t = g_{\boldsymbol{\theta},\boldsymbol{\alpha}}\left(x_t, h_{t-1}, c_{t-1}\right) , \tag{5}$$

where $\boldsymbol{\theta}$ are the trainable parameters, $\boldsymbol{\alpha}$ is the structure of g, h_t is the hidden state, c_t the cell state, and x_t the input at step t. The search space defines all its elements $\boldsymbol{\alpha}$ by a full binary tree where every node is associated with a binary merge operation (e.g. addition, element-wise multiplication) and an activation function (e.g. tanh, sigmoid) as shown in Figure 8. Every node i represents a parameterized function

$$g_{\boldsymbol{\theta}}^{(i)}\left(a_1, a_2\right) = o^{(\mathrm{act})}\left(o^{(\mathrm{merge})}\left(a_1, a_2\right)\right) , \tag{6}$$

where a_1, a_2 is the output computed by the child nodes of node i. Each leaf node takes as input x_t, h_{t-1}, the cell state is considered independently. A node within the tree has to be selected to provide c_t. c_{t-1} is used in combination with an arbitrary node using the selected merge operation and activation function.

3. Optimization Methods

In the following, we formally define the problem of neural architecture search. We denote the space of all datasets as D and the space of all deep learning models as M. Furthermore,

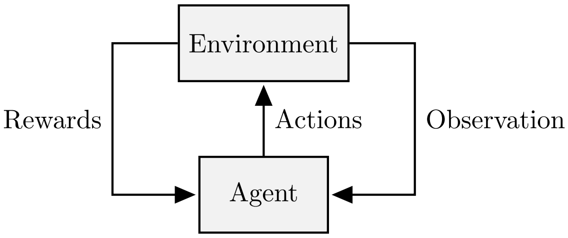


Figure 9: A general framework for reinforcement learning algorithms.

we denote the architecture search space as A. Then, a general deep learning algorithm Λ is a mapping

$$\Lambda \ : \ D \times A \rightarrow M \ . \tag{7}$$

An architecture of the search space $\boldsymbol{\alpha} \in A$ defines more than just the topology. It further encodes all properties to train the architecture on a dataset which includes, e.g. the optimizer choice, regularization and all other hyperparameters.

Given a dataset d, which is split into a training partition d_{train} and a validation partition d_{valid}, the general deep learning algorithm Λ estimates the model $m_{\boldsymbol{\alpha},\boldsymbol{\theta}} \in M_{\boldsymbol{\alpha}}$. This model is estimated by minimizing a loss function $\mathcal{L}$ which is penalized with a regularization term $\mathcal{R}$ with respect to the training data. That is,

$$\Lambda\left(\boldsymbol{\alpha}, d\right) = \underset{m_{\boldsymbol{\alpha},\boldsymbol{\theta}} \in M_{\boldsymbol{\alpha}}}{\arg\min} \ \mathcal{L}\left(m_{\boldsymbol{\alpha},\boldsymbol{\theta}}, d_{\text{train}}\right) + \mathcal{R}\left(\boldsymbol{\theta}\right) \ . \tag{8}$$

Neural architecture search is the task of finding the architecture α^* which maximizes an objective function $\mathcal{O}$ on the validation partition d_{valid}. Formally,

$$\boldsymbol{\alpha}^* = \underset{\boldsymbol{\alpha} \in A}{\arg\max} \ \mathcal{O}\left(\Lambda\left(\boldsymbol{\alpha}, d_{\text{train}}\right), d_{\text{valid}}\right) = \underset{\boldsymbol{\alpha} \in A}{\arg\max} \, f\left(\boldsymbol{\alpha}\right) \ . \tag{9}$$

The objective function $\mathcal{O}$ can be the same as the negative loss function $\mathcal{L}$. For the classification problem it is often the case that the loss is the negative cross-entropy and the objective function the classification accuracy.

Optimizing the *response function* f is a global black-box optimization problem. In the following, we discuss several optimization strategies based on reinforcement learning, evolutionary algorithms and others.

3.1 Reinforcement Learning

Reinforcement Learning approaches are useful for modeling a process of sequential decision making where an agent interacts with an environment with the sole objective of maximizing its future return. The agents learns to improve its behavior through multiple episodes of interaction with the environment. At every step, the agent executes an action and receives an observation of the state along with a reward. The environment on the other hand receives the agent's action, accordingly transits to a new state and emits the corresponding observation and reward (Figure 9). The objective of the agent lies in maximizing its return which is often a discounted sum of rewards. This formalism fundamentally relies on the concept of a state which can be thought of as a sufficient statistic for the history of agent or environment, i.e. the agent's future behaviour depends entirely on the current state. Naturally, such approaches are well suited for neural architecture search where the agent, namely the search algorithm, takes decisions to modify the system's state, i.e. the architecture, so as to maximize the long term objective of high performance which is often measured in terms of accuracy.

Reinforcement learning (RL) systems are often modeled as a *Markov decision process* encompassing cases where the environment is fully observable. Most approaches consider simplified scenarios where the set of actions and states is finite and the setup consists in a finite horizon problem where episodes terminate after a finite number of steps. Even in this setting the number of possible options is combinatorially high. An RL agent seeks to learn a policy π which serves as the mapping from a state to a probability distribution over the set of actions. While one class of RL methods indirectly learn this policy with the use of value functions and state-action value functions, other approaches directly learn a parameterized policy. We formalize these two approaches in the following text.

An RL system can be more formally described as follows: at every decision step t, the agent takes an action $a^{(t)}$ to modify the current state $s^{(t)}$ and receives a reward $r^{(t)}$. The agent's sole objective is to maximize its cumulative reward or return over T decision steps as defined by $\sum_{t=0}^{T} \gamma r^{(t)}$, where γ is the discounting factor. The state-value function $v_\pi(s)$ defines the expected return for the agent given that the starting state is s and the policy π is followed thereafter. The value of taking an action a in state s under a policy π is similarly defined by q_π. Both value functions satisfy the consistency equation defined by the Bellman equation (Sutton and Barto, 1998). Furthermore, the value function induces a partial order over policies and the optimal policy is the one that ascribes largest values to all states. The goal of an agent is to find this optimal policy.

Temporal Difference Learning Approaches like SARSA, TD-λ, and Q-learning attempt to find this policy implicitly via approximating the optimal value functions. The optimal policy is subsequently determined as the greedy-policy with respect to the optimal value function. The optimal value functions $v^*(s)$ and $q^*(a, s)$ satisfy the Bellman optimal criterion. Q-learning (Watkins, 1989) learns an action-value function $Q(s, a)$ that directly approximates q^*. The agent learns Q independently from the policy it follows. The Q-learning algorithm uses the following update rule for the action-value function Q,

$$Q(s^{(t)}, a^{(t)}) \leftarrow (1-\eta)\, Q(s^{(t)}, a^{(t)}) + \eta \left[r^{(t)} + \gamma \max_{a'} Q\left(s^{(t+1)}, a'\right) \right] . \tag{10}$$

Here, η is the Q-learning rate. The learned policy and correspondingly the proposed architecture is subsequently derived by greedily selecting the actions for every state which maximize $Q(s, a)$.

Policy Gradient Methods Other alternate approaches in RL, collectively referred to as policy gradient methods, do not appeal to value functions and instead directly learn policies as defined by a collection of parameters, $\pi_{\boldsymbol{\theta}}(a|s)$. These methods select actions without explicitly consulting a value function. The parameters of the policy $\boldsymbol{\theta}$ are adjusted so as to move in the direction of the agent's performance measure via classical gradient ascent updates The required gradient is often not directly available as the performance depends both on the actions selected and the distribution of states under which the actions

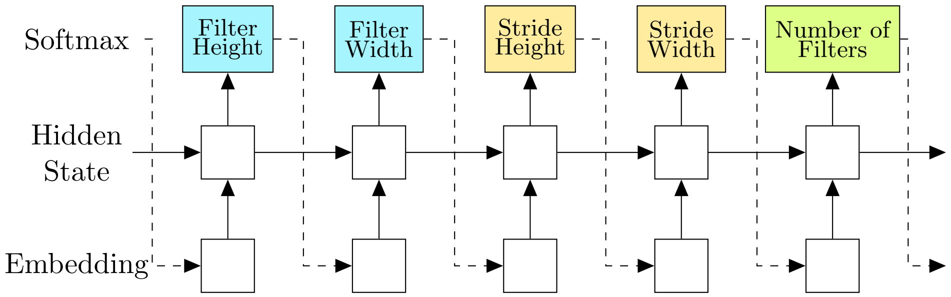


Figure 10: The controller used by Zoph and Le (2017) (predictions for skip connections are not shown) to predict configuration of one layer.

is taken. Hence an empirical estimate of the gradient is used to perform the necessary update. REINFORCE (Williams, 1992) is a classical algorithm that estimates this gradient as,

$$\mathbb{E}_{\pi}\left[\sum_{t=0}^{T}\nabla_{\boldsymbol{\theta}}\ln\pi_{\boldsymbol{\theta}}(a^{(t)}|s^{(t)})G^{(t)}\right] . \qquad (11)$$

where $G^{(t)}$ is the return from step t, $\left(\sum_{t}^{T}\gamma r^{(t)}\right)$. However these empirical gradient estimates are often noisy and additional tricks like inclusion of a baseline are incorporated for useful learning. Moreover, alternative formulations of this gradient as derived through objective functions of importance sampling have led to other approximations. These approaches include Trust Region Policy Optimization (TRPO) (Schulman et al., 2015) and Proximal Policy Optimization (PPO) (Schulman et al., 2017). It is worth noting that these methods are on-policy methods as the agents follow the policy they seek to learn.

Optimization with Q-Learning Baker et al. (2017) were one of the first to propose the use of RL-based algorithms for neural architecture search. They use a combination of Q-learning, ϵ-greedy, and experience replay in the design of their algorithm. The actions in their approach are the choice of different layers to add to an architecture as well as the option to terminate building the architecture and declare it as finished (we discussed this search space in Section 2.1). Subsequently, the states are the premature architectures. The trajectories sampled from this state space correspond to models which are subsequently trained to compute the validation accuracy. The Q function is appropriately updated with experience replay. In order to trade off between exploration and exploitation, they incorporate an ϵ-greedy strategy where random trajectories are sampled with a probability of ϵ. This is an off-policy method where the agent does not use the optimal policy during the episodes. Additionally, the action-value function utilized in this case is deterministic. The algorithm works as follows: after initializing the action-value function, the agent samples a trajectory which comprises of multiple decision steps, eventually leading to a terminal state. The algorithm then trains the model corresponding to the trajectory and updates the action-value function as defined in the Q-learning algorithm. Zhong et al. (2018) also use a Q-learning-based algorithm to search for network architectures. However, they perform the search in a cell-based search space which we discussed in Section 2.2.

Optimization with Policy Gradient Methods Alternate approaches based on policy gradient method have also been used for neural architecture search. The work by Zoph and Le (2017) was the first to consider this modeling approach. In their approach they directly model a controller whose predictions can be considered as actions that are used to construct a neural architecture. The controller parameterized by $\boldsymbol{\theta}$ defines the stochastic policy $\pi_{\boldsymbol{\theta}}(a|s)$. They incorporate an autoregressive controller which predicts the action based on previous actions, and model it with a Recurrent Neural Network (RNN). During an episode, every action is sampled from the probability distribution implied by a softmax operation and then fed into the next time step as input (Figure 10). This RNN-controller in their approach samples layers which are sequentially appended to construct the final network. The final network is trained to obtain the validation accuracy and the parameters of the controller are updated as per the REINFORCE update rule (Equation (11)) which involves scaling the gradient with the obtained validation accuracy. As discussed in Section 2.1, they consider a space of sequential networks parameterized by only convolutional layers with different filter width, filter height, stride parameters, and number of filters. Additionally, they include anchor points in their encoding to allow for skip connections.

Policy gradient approaches have also been explored to learn controllers for cell-based search spaces. Zoph et al. (2018) define an RNN-controller that outputs actions which sequentially determines the inputs and operations for a prespecified number of blocks within a NASNet search space cell as defined in Section 2.2. The sampled architectures are trained and the parameters of the controller are updated according to the PPO update rule.

Cai et al. (2018a) also model a controller to parameterized the policy and train it with REINFORCE. However the action space of the controller differs and it requires an initial architecture to start with. There are two types of actions, widening a layer or inserting a new layer to deepen the network. Both actions are performed by means of *function-preserving transformations* (Chen et al., 2016). This means that although the structure of the architecture changes, the function that it models remains unchanged. Thus, the accuracy of this network does not change at first. This is ensured by initializing the additionally added parameters in a special way. In the example of adding a layer, it is initialized to correspond to the identity function. At every step and every layer, the controller predicts whether it wants to change it by means of widening or deepening it. As this involves predicting transformation decisions for a variable length sequence, they incorporate a bidirectional recurrent network (Schuster and Paliwal, 1997) with an embedding layer to learn a low dimensional representation of the network which is subsequently fed into actors that compute the transformation decisions. For the predicted actions the function-preserving transformations are computed which enable efficient reuse of weights for training the newly sampled architecture which reduces the overall search duration.

The work by Cai et al. (2018b) builds upon these transformations and propose new transformations to include branches in the network architectures. A convolutional layer or an identity map is first converted to its equivalent multi-branch motif such that the functionality is preserved. This is achieved by splitting or replicating the input to a convolutional or identity map into multiple branches, applying the corresponding transformation (convolutional or identity) to each branch, and respectively concatenating or adding the outputs of each branch. The function-preserving transformations are subsequently applied on branches to explore more complex architectures. Similar to the work by Cai et al.

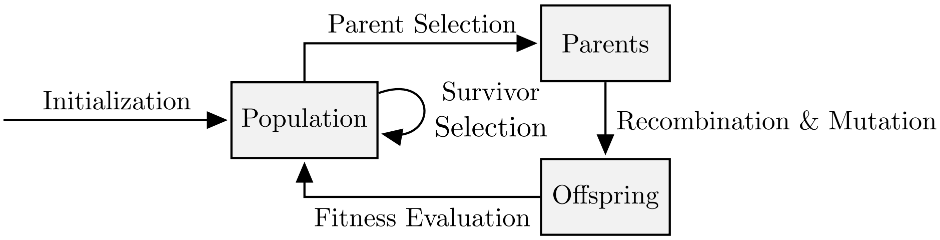


Figure 11: A general framework for evolutionary algorithms.

(2018a), the meta-controller in their approach also uses an encoder network to learn a low-dimensional representation of the architecture and provide a distribution over transformation actions. They propose to learn the meta-controller with REINFORCE.

3.2 Evolutionary Algorithms

Evolutionary algorithms (EA) are population-based global optimizer for black-box functions which consist of following essential components: initialization, parent selection, recombination and mutation, survivor selection. The initialization defines how the first generation of the population is generated. After the initialization the optimizer repeats the following steps until termination (see Figure 11):

1. Select parents from the population for reproduction.
2. Apply recombination and mutation operations to create new individuals.
3. Evaluate the fitness of the new individuals.
4. Select the survivors of the population.

In this broad class of algorithms, the choices for mutation and recombination operators along with the choice of fitness and parent selection functions guide the overall search process. The choice of operators used for recombination and mutation is motivated to trade off diversity and similarity in the population, akin to the exploration and exploitation trade-off in reinforcement learning-based search algorithms. Similarly, the choice of fitness functions reflects the optimization objective and the choice of survivor selection enables competition between the individuals of the population.

In the context of neural architecture search, the population consists of a pool of network architectures. A parent architecture or a pair of architectures is selected in step 1 for mutation or recombination, respectively. The steps of mutation and recombination refer to operations that lead to novel architectures in the search space which are evaluated for fitness in step 3 and the process is repeated till termination. Often only mutation operators are used in the domain of neural architecture search. There is no indication that a recombination operation applied to two individuals with high fitness would result into an offspring with similar or better fitness. On the contrary, oftentimes the fitness of the resultant offspring is much lower. For this reason and other reasons of simplicity, this part is often omitted.

The most common parent selection method in neural architecture search is *tournament selection* (Goldberg and Deb, 1990). This method selects the individuals for further evolution in two steps. First, it selects k individuals from the population at random. Then

Table 1: High-level details of various evolutionary algorithms for neural architecture search.

Method	Search Space	Init	Parent Sel.	Survivor Sel.
Real et al. (2017)	global	simple	tournament	tournament
Xie and Yuille (2017)	global	random	all	fitness prop.
Suganuma et al. (2017)	global	random	n/a	elitist
Liu et al. (2018b)	cell-based	random	tournament	all
Real et al. (2019)	cell-based	random	tournament	youngest
Elsken et al. (2018)	global	simple	n/a	elitist
Wistuba (2018)	cell-based	simple	tournament	all

it iterates over them in the descending order of their fitness while selecting individuals for further steps with some (prespecified) high probability p. An alternative to tournament selection is *fitness proportionate selection*. In this approach an individual is selected proportional to its fitness. Thus, in a population $\{\alpha_1, \ldots, \alpha_N\}$, the i-th individual is selected with probability $\frac{f(\alpha_i)}{\sum_{j=1}^{N} f(\alpha_j)}$, where $f(\alpha_i)$ is the fitness of the individual α_i.

After the recombination and mutation step, the population has grown. The intent of the survivor selection component is to reduce the population size and enable competition within the individuals. Several different policies are used to achieve this, ranging from selecting only the best (*elitist selection*) to selecting all individuals. One class of evolutionary algorithm that has been widely adopted for neural architecture search is genetic algorithms. These approaches bear their name from the representation of individuals in their methodology which is done by means of a fixed-length encoding, called the *genotype*. Mutation and recombination operations are performed directly on this representation. Further, a genotype is used to create the physical appearance of the individual, which in the case of neural architecture search is the architecture. This materialization is called the *phenotype*. It is important to note that parts of the genotype can be conditionally active or inactive. The information of the genotype is referred to as *active* if its modification results in a change in the phenotype (given all other information remains the same). For an example see Figure 13.

In the remaining section we discuss the popular choices for search space, mutation operators and selection functions that have been utilized for neural architecture search. We note that EA-based neural architecture search methods include a set of highly diverse approaches which have benefited from the varied choices of encoding the search space along with the choices of mutation operators and selection functions. In the context of this work we describe six notable works in EA-based neural architecture search. A broad overview of these approaches is provided in Table 1.

Real et al. (2017) Real et al. (2017) were one of the first to propose an evolutionary algorithm to find competitive convolutional neural network architectures for image classification. Their approach begins with one thousand copies of the simplest possible architecture, i.e. a global pooling layer followed by the output layer. In the parent selection step of their approach they propose to sample a pair of architectures for further processing. While the better of the two architectures is copied (along with the weights), mutated, trained for

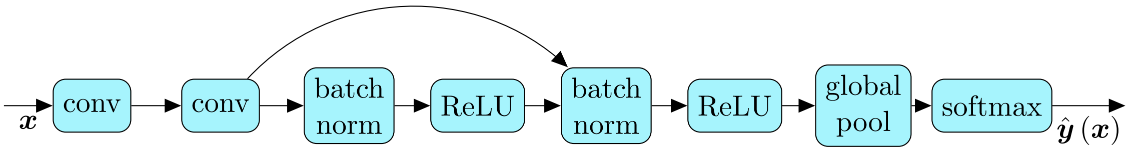


Figure 12: A possible architecture discovered by the algorithm described by Real et al. (2017). Noticeable are the redundant operations such as two convolutions in a row without an activation.

25,600 steps with a batch size 50, and added to the population, the other one is removed from the population. This is equivalent to using tournament selection with $k = 2$ and $p = 1$. The set of mutations consists of simple operations such as adding convolutions (possibly with batch normalization or ReLU activation) at arbitrary locations before the global pooling layer, along with mutations that alter kernel size, number of channels, stride and learning rate, add or remove skip connections, remove convolutions, reset weights, and a mutation that corresponds to no change. Since no further domain constraints are enforced, architectures with redundant components can be potentially sampled in this approach (see Figure 12).

Xie and Yuille (2017) Xie and Yuille (2017) made their work available only one day later. In contrast to Real et al. (2017), their work develops a genetic algorithm over a more structured search space which comprises of a sequence of segments with possibly parallel convolution operations. In their approach they consider networks to be composed of three segments, each consisting of multiple convolution layers (see Section 2.1 for more details). Each segment is described by an adjacency matrix which defines a directed acyclic graph over the layers within the segment. The genotype of the entire network is obtained as a fixed length binary string of the adjacency matrices of the three segments. The algorithm begins with a population of 20 random samples from the search space. In the parent selection step, all pairs $(\alpha_i, \alpha_{i+1})\,,\ i \mod 2 = 1$ are considered for a cross-over operation. With a probability p, the cross-over operation swaps segments between the two selected networks. As the next step, all individuals which have undergone no modifications in the previous step are considered for mutation. In their algorithm, mutations are defined as random flip operations on the adjacency matrices that define the segments. Finally, the obtained offspring is trained from scratch and its fitness is evaluated. The fitness of an individual is defined as the difference between its validation accuracy and the minimum accuracy among all individuals of the population. As fitness proportionate selection is used in their approach, this modification ensures that the weakest individual survives with a probability of zero. It is worth remarking that this is one of the first works to demonstrate a successful transfer of an architecture automatically discovered on a smaller dataset to a larger one, namely from CIFAR-10 to ImageNet (Krizhevsky et al., 2012).

Suganuma et al. (2017) Suganuma et al. (2017) present another optimizer based on genetic algorithms but consider a wider set of operations in their search space which includes convolutions and pooling along with concatenation and summation of vectors. In their

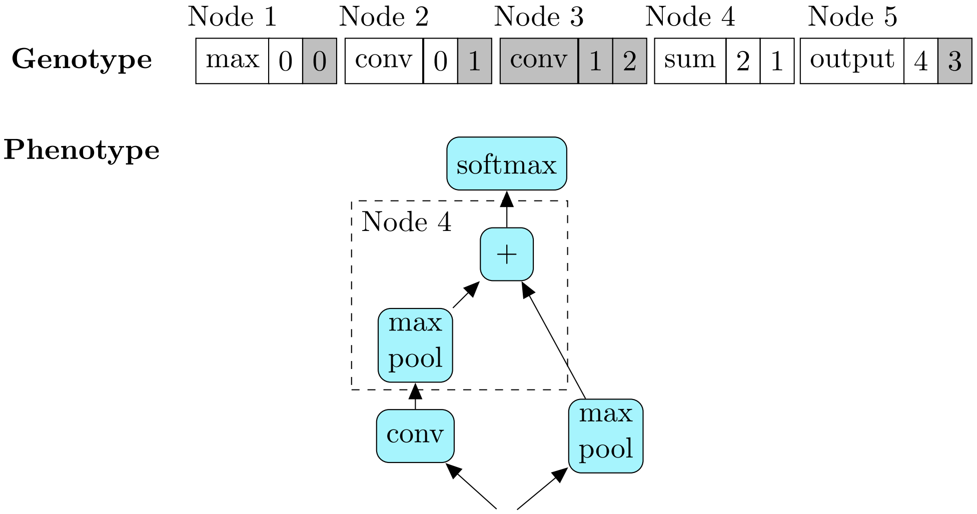


Figure 13: The genetic encoding used by Suganuma et al. (2017). Gray shaded parts of the genotype are inactive. Inactive parts such as Node 3 are not present in the phenotype. The additional max pooling layer introduced by Node 4 is an example how not explicitly encoded operations in the genotype can appear in the phenotype.

approach, they encode the entire network as a sequence of blocks represented by a triple that defines the operation and inputs of the block. The string obtained by concatenating the block encoding constitutes the genotype. Inactive parts are admissible in such a definition of a genotype, for instance unused inputs for certain operations or disconnected blocks. The inactive parts do not materialize in the phenotype (for an example see Figure 13). They propose to use the (1+λ)-evolutionary strategy ($\lambda = 2$) (Beyer and Schwefel, 2002) approach to guide the evolution where mutation and selection operators are applied in a loop until termination. Beginning with a genotype that corresponds to the parent individual (random for the first iteration), λ offspring are generated by forcefully modifying the active parts of the genotype. A mutation is also applied to the parent on one of the inactive parts of its genotype. However this does not result in any change of the phenotype and hence does not add to the overall training cost. This action ensures progress of the search even in cases where the parent of this generation will become the parent of the next generations again. The offspring networks are trained to obtain validation accuracies and the parent for the next iteration is selected using elitist selection strategy.

Liu et al. (2018b) Liu et al. (2018b) propose another evolutionary algorithm which uses the hierarchically organized search space as described in Section 2.2. At every step a mutation selects the hierarchy level to modify along with the modification to the representations at that level. The population is initialized with 200 trivial genotypes which are diversified by applying 1000 random mutations. Furthermore, parents are selected using tournament selection (5% of the population) and no individuals are removed during the evolutionary process. In their setting, mutations can add, alter and remove operations from the genotype.

Real et al. (2019) The follow-up work by Real et al. (2019) is one of the most significant works in the direction of using evolutionary algorithms for architecture search. It is primarily known for finding the architectures AmoebaNet-B and AmoebaNet-C which set new records for the task of image classification on CIFAR-10 and ImageNet dataset. However, their search process used a total of 3,150 GPU hours. Their evolutionary search algorithm operates on the NASNet search space (details in Section 2.2) with tournament selection used to select parents at each iteration. The selected individuals are mutated with a random change in an operation or a connection in either a normal or a reduction cell and are trained for 25 epochs. As opposed to previous works, their algorithm does not solely rely on validation accuracy and incorporates age in the selection of survivors. This is motivated to restrain repeated selection of well-performing models for mutation and introduce diversity to the population. This basically adds a regularization term to the objective function which makes sure that we are searching for architectures which are not only capable of reaching high validation accuracy once but every time.

The previously discussed evolutionary algorithms are mostly concerned about finding a good performing architecture ignoring GPU budget limitations. The shortest search time used among those methods is still seventeen GPU days. The two works discussed in the following are more concerned about efficiency and report comparable results within a day. The method to accomplish this is a combination of mutations which are function-preserving transformations (see Section 3.1) and a more aggressive, greedy evolution.

Elsken et al. (2018) Elsken et al. (2018) propose a simple yet efficient evolutionary algorithm inspired by the function-preserving transformations. Similar to Suganuma et al. (2017) they follow the (1+λ) evolutionary strategy. A parent is selected with elitist selection strategy, eight different offspring are generated with function-preserving mutations, and each is trained for a total of 17 epochs. The initial architecture comprises of three convolution and two max pooling layers. As discussed earlier, the function-preserving operations significantly reduce the training time per architecture and therefore the entire search duration.

Wistuba (2018) In a similar line of work, Wistuba (2018) also utilizes function-preserving mutations but uses a cell-based search space. The proposed search method begins with a simple network and performs function-preserving mutations to grow the population. The initial population is diversified by generating fifteen additional children from the base network. During the evolutionary algorithm, parents are selected using tournament selection and the offspring is trained for fifteen epochs prior to their fitness evaluation. Similar to Liu et al. (2018b), all individuals survive the evolution process to ensure diversity. Discovered cells have arbitrary structures, an example is provided in Figure 14.

3.3 Surrogate Model-Based Optimization

As the name implies, surrogate model-based optimizers use a surrogate model $\hat{f}$ to approximate the response function f (Equation (8)). For the case of neural architecture search this is motivated to get an approximate response for an architecture without the time-consuming training step and improve the overall efficiency of the search process. The surrogate is modeled as a machine learning model and is trained on a meta-dataset which

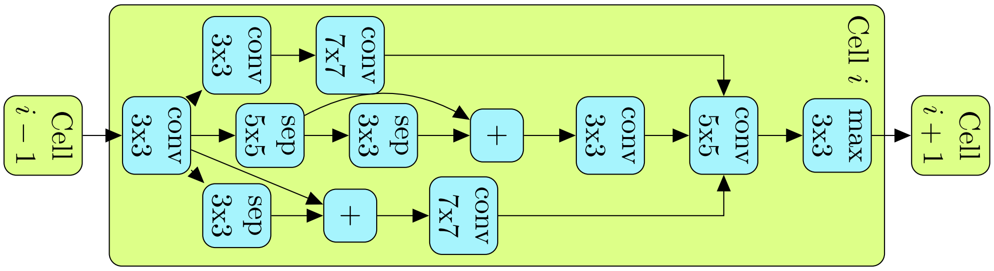


Figure 14: Exemplary outcome of the evolutionary algorithm proposed by Wistuba (2018).

contains architecture descriptions with their corresponding values of the response function, which are gathered during the architecture search:

$$H = \left\{ \left(\boldsymbol{\alpha}_1, f\left(\boldsymbol{\alpha}_1,\right)\right), \left(\boldsymbol{\alpha}_2, f\left(\boldsymbol{\alpha}_2,\right)\right), \ldots \right\} \tag{12}$$

Surrogate models are generally trained to minimize the squared error:

$$\sum_{(\boldsymbol{\alpha}, f(\boldsymbol{\alpha})) \in H} \left(\hat{f}\left(\boldsymbol{\alpha}\right) - f\left(\boldsymbol{\alpha}\right)\right)^2 . \tag{13}$$

However, sometimes in practice only a ranking for the architecture is desired and in such cases a small test error is not necessitated as long as the surrogates provide a useful ranking.

The predictions from surrogate model are often used to identify promising candidate architectures. These candidates are evaluated, their corresponding new meta-instances are added to H and the surrogate model is accordingly updated. These steps are executed until a convergence criterion is reached. We describe three different approaches to surrogate model-based optimization that have been used with an intent to improve the efficiency of neural architecture search.

Kandasamy et al. (2018) cast this as a black-box optimization problem and tackle it with Bayesian optimization (Snoek et al., 2012). Bayesian optimization uses a probabilistic surrogate model and an acquisition function which measures the utility by accounting for both, the predicted response and the uncertainty in the prediction. In their approach they model the surrogate with a Gaussian process (Rasmussen and Williams, 2006) and use expected improvement as the acquisition function (Mockus et al., 1978) for the optimization. The architecture with the highest expected utility is selected for evaluation and the consequent meta-instances are added to H. The surrogate model is updated and the previous steps are repeated. The authors render two changes to the standard Bayesian optimization algorithm. A novel kernel function to compute similarity between two network architectures is proposed as well as an evolutionary algorithm to maximize the acquisition function. The kernel value computation in this work is modeled as an optimal transport program (Villani, 2008). In essence, the graph topology along with the location and frequency of operations as well as the number of feature maps is used to compute the similarity between two architectures.

Liu et al. (2018a) also incorporate a surrogate model and search for architectures in the NASNet search space (Section 2.2). They explore this search space by progressively increasing the number of blocks in a cell, which, as previously discussed, is a hyperparameter

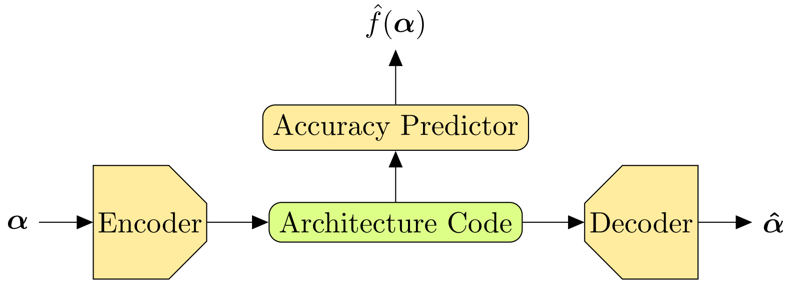


Figure 15: Luo et al. (2018) propose to combine an autoencoder with the surrogate model. This model is jointly learned to achieve $\boldsymbol{\alpha} \approx \hat{\boldsymbol{\alpha}}$ and $f(\boldsymbol{\alpha}) \approx \hat{f}(\boldsymbol{\alpha})$.

and fixed in the template. While searching for models with an additional block, they simultaneously train a surrogate model which can predict the validation accuracy of the architecture given the encoding of the cell. The overall search process is designed as a beam search where at every step, a new set of candidates is generated by expanding the current pool of cells with an additional block and the top k of those are trained as per the predictions of the surrogate model. Further, in each step the surrogate model is updated by considering the newly obtained k architectures in the meta-dataset. Given the design of their setup, the surrogate model is required to handle variable sized inputs. In addition to an RNN and an Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) which are natural choices for this modeling, they also investigate a multi-layer perceptron (MLP) for this model. With blocks being completely defined by the pair of input nodes and the corresponding operation, they learn embedding vectors for the set of input nodes and operations. For the case of the MLP surrogate, the block embedding is defined as the concatenation of these vectors and the cell embedding as the mean embedding of its blocks.

In an interesting approach, Luo et al. (2018) jointly learn an autoencoder for the architecture representation with the surrogate model which uses the continuous encoding provided by the autoencoder, the architecture code, as its input (Figure 15). A key difference lies in their search algorithm which uses the surrogate model to sample new architectures by taking gradient steps with respect to the architecture code. The architecture code is changed by moving in the direction which yields better accuracy according to the surrogate model. The new architecture is obtained by mapping the potentially better architecture code back using the decoder learned as part of the autoencoder. At every step, these samples are trained, the meta-dataset is expanded, and the surrogate model and autoencoder are updated accordingly. The encoder, decoder and the surrogate model are jointly trained to minimize a weighted sum of the reconstruction loss and the performance prediction loss.

Experiment The effectiveness of surrogate model-based optimization methods often critically depends on how correlated the response values obtained from the surrogate model are to the ones obtained for the true response function. We verify this underlying assumption for the work of Luo et al. (2018). We trained 650 random architectures on CIFAR-10 for 100 epochs and estimate their validation accuracy. Of these, 50 instances are reserved for evaluating the surrogate model. We observe that the correlation between predicted and true validation accuracy of these 50 instances is gradually increasing with a growing number of

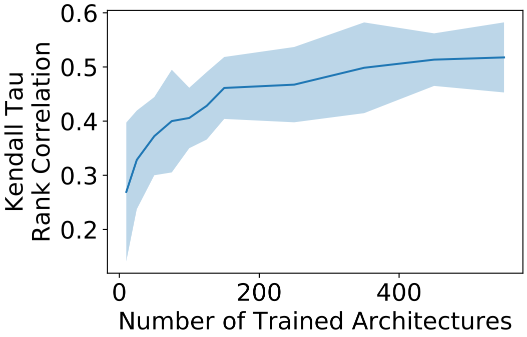


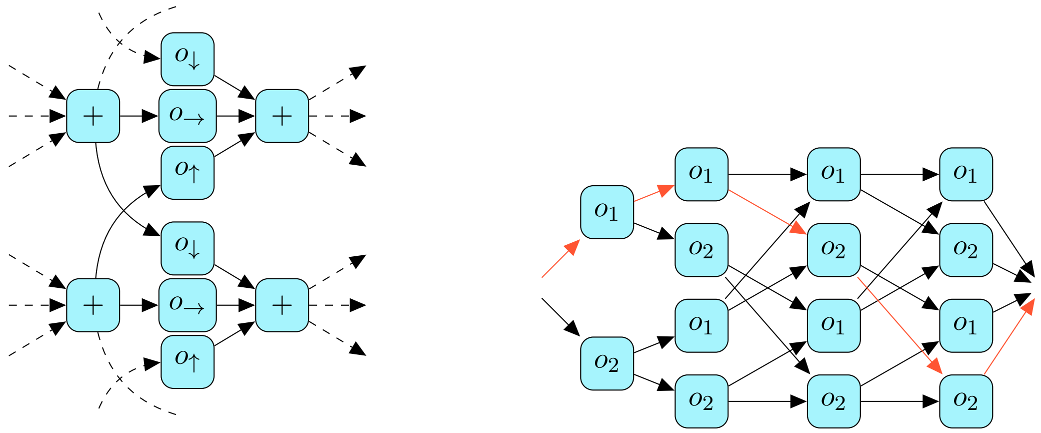
Figure 16: The ranking of architectures implied by surrogate models show positive correlation with the true architecture ranking.

training architectures. At every step, we repeat the experiment ten times with a different sample for the meta-dataset and report the mean and standard deviation in Figure 16. Even with a surrogate model learned on only 10 meta-instances, we notice a high positive correlation. Thus, we can conclude that the surrogate model is able to provide a better than random ranking of architectures.

3.4 One-Shot Architecture Search

We define an architecture search method as one-shot if it trains a single neural network during the search process. This neural network is then used to derive an architecture throughout the search space as a solution to the optimization problem. Most architectures considered by one-shot methods are based on an over-parameterized network (Saxena and Verbeek, 2016; Pham et al., 2018; Liu et al., 2018c; Xie et al., 2019b; Casale et al., 2019). The advantage of this family of methods is the relatively low search effort which is only slightly greater than the training costs of one architecture in the search space. As we describe later, this methodology can be combined with many of the previously discussed optimization methods.

Saxena and Verbeek (2016) are the first to use this methodology, but their approach differs from the later works in the design of their search space. The two major differences are that only convolutions operations are considered and the architecture $\boldsymbol{\alpha}$ is fixed. The architecture has arbitrary depth as determined by a hyperparameter and a set number of levels which depends on the size of the input. Each level has a resolution of a power of two from 1 to the input size. For each depth three types of operations are considered: one that reduces the resolution of the higher level by a factor of two, one that increases the resolution of the lower level by a factor of two, and one that retains the resolution unchanged. The output of these three operations is added together and forms the output of the level at that depth (see Figure 17a). This architecture is trained and ultimately forms the final model. In a post-processing step, individual paths can be removed to reduce the number of network parameters at marginal cost for accuracy. However this method requires a total number of model parameters which is several times higher than other architectures which obtain a similar accuracy.



(a) Convolutional Neural Fabric

(b) Weight Sharing

Figure 17: Left: A part of the convolutional neural fabric at a specific depth. Operations for two different shape sizes are shown (e.g. 4 and 8). Convolutional neural fabrics can have arbitrary depth. Right: Example for weight sharing with only two operations in the sequential search space. Every box has its own weights, every path (e.g. the red one: $o_1 \rightarrow o_1 \rightarrow o_2 \rightarrow o_2$) is one architecture in the search space. Thus, weights are shared across different architectures.

Weight Sharing Pham et al. (2018) search in a subspace of the NASNet search space (see Section 2.2) and operate on an over-parameterized network that covers the entire search space. In contrast to the work of Saxena and Verbeek (2016), the complete architecture is not taken into account in every training step. Instead, Pham et al. (2018) use reinforcement learning to learn a controller to sample architectures (which are only subgraphs of the complete architecture) from the search space (see Figure 17b). Alternately, the weights of the controller and a part of the over-parameterized network will be updated by gradient descent. During the network parameter update step, a batch is selected and an architecture is sampled from the search space. Only the parameters of this architecture are updated, the remaining parameters and the parameters of the controller remain unchanged. During the update of the controller, architectures are sampled and evaluated on a validation batch. The reward determined in this way is used to update the controller similar to Zoph and Le (2017) (details provided in Section 3.1). This idea is referred to as weight sharing, as it corresponds to training multiple networks with shared weights. Bender et al. (2018) suggest replacing the controller with uniform sampling. Casale et al. (2019) learn the parameters of an independent categorical distribution to sample architectures. In all these variations, after the training, the final architecture is selected by means of the respective sampling and evaluation strategy and trained from scratch.

Differentiable Architecture Search Let us recall the formal definition of a neural network:

$$\boldsymbol{z}^{(k)} = \sum_{j \in |O|} o^{(j)} \left(\left\{ \boldsymbol{z}^{(i)} \mid \alpha_{i,j,k} = 1,\ i < k \right\} \right) \tag{14}$$

For notational convenience, we will assume that each operation which determines $\boldsymbol{z}^{(k)}$ can use only a single input from $\mathcal{I}^{(k)}$. Then, this definition simplifies to

$$\boldsymbol{z}^{(k)} = \sum_{i \in \mathcal{I}^{(k)}} \sum_{j \in |O|} \alpha_{i,j,k} \cdot o^{(j)} \left(\left\{ \boldsymbol{z}^{(i)} \right\} \right) , \tag{15}$$

with $\alpha_{i,j,k} \in \{0, 1\}$. So far the assumption has been that every operation is either part of the network or not. Liu et al. (2018c) relaxes this assumption and instead assumes a linearly weighted combination where $\alpha_{i,j,k}$ can assume any real value in the range of 0 to 1. They parameterize $\boldsymbol{\alpha}$ by $\boldsymbol{\beta}$,

$$\alpha_{i,j,k} = \frac{\exp\left(\beta_{i,j,k}\right)}{\sum_{i \in \mathcal{I}^{(k)}} \sum_{j \in |O|} \exp\left(\beta_{i',j',k}\right)} . \tag{16}$$

This softmax operation makes sure for every k that

$$\sum_{i \in \mathcal{I}^{(k)}} \sum_{j \in |O|} \alpha_{i,j,k} = 1 . \tag{17}$$

From Equation (9) a new, differentiable loss function for both structural parameters and model parameters can be derived:

$$\min_{\boldsymbol{\alpha}(\boldsymbol{\beta}) \in A} \mathcal{L} \left(\operatorname*{arg\,min}_{m_{\boldsymbol{\alpha}(\boldsymbol{\beta}),\boldsymbol{\theta}} \in M_{\boldsymbol{\alpha}(\boldsymbol{\beta})}} \mathcal{L}\left(m_{\boldsymbol{\alpha}(\boldsymbol{\beta}),\boldsymbol{\theta}}, d_{\text{train}}\right) + \mathcal{R}\left(\boldsymbol{\theta}\right), d_{\text{valid}} \right) . \tag{18}$$

Liu et al. (2018c) propose an alternating optimization method which learns the parameters of the model $\boldsymbol{\theta}$ by minimizing the loss on the training set and the structural parameters $\boldsymbol{\beta}$ by minimizing the loss on the validation set using gradient-based optimization methods. After training this network, the final architecture is chosen based on the values of $\alpha_{i,j,k}$, the larger the better. This method is a very elegant solution that makes all parameters differentiable for both the model and the architecture. However, it has a significant drawback: all parameters must be kept in memory all the time. Since the network covers the entire search space, this is a significant disadvantage. The following two works overcome this by introducing update rules which require to keep only a part of the network in memory at any one time.

The work by Xie et al. (2019b) is based on the achievements of Liu et al. (2018c), but has three major differences. First, they represent the architecture by $p\left(\boldsymbol{\beta}\right)$, where $p\left(\boldsymbol{\beta}\right)$ is fully factorizable and modeled by a concrete distribution (Maddison et al., 2017). Choosing this distribution causes only one path to be selected in the over-parameterized network. This path corresponds to a feasible solution in the search space and accordingly requires less memory and can be easily kept in memory. Second, they minimize the loss of d_{train} with respect to the two parameters $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$. The validation partition d_{valid} is used only to select the final architecture. And thirdly, gradients are estimated from Monte Carlo samples rather than analytic expectation.

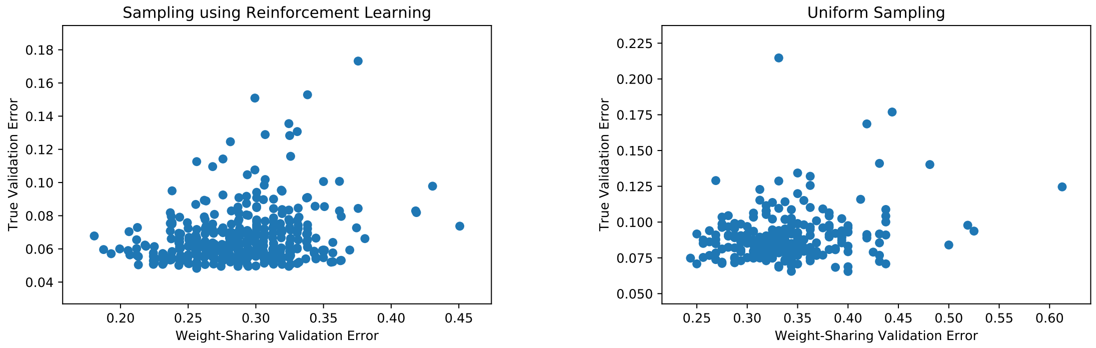
Cai et al. (2019) also tries to reduce the memory overhead of differentiable architecture searches by always having to load only one path of the over-parameterized network into the memory at any one time. To do this, they use binary gates (Courbariaux et al., 2015) that model the structural parameters of the over-parameterized network. Each binary gate chooses a part of the path based on a learned probability and in this sense is very similar to the idea of Xie et al. (2019b).

Discrete architectures are derived by selecting the most likely operation and the top two predecessors in the over-parameterized network. This architecture is then trained again from scratch.

Hypernetworks Brock et al. (2018) propose the use of dynamic hypernetworks (Ha et al., 2017), a neural network which generates the weights for another network conditioned on a variable, i.e. in this case the architecture description. The hypernetwork is trained to generate network weights for a variety of architectures. It can be used to rank different architectures and derive a final architecture which is then trained from scratch. This method also shares weights, but most of them are shared in the hypernetwork.

Zhang et al. (2019) extend this idea by combining it with a graph neural network (Scarselli et al., 2009). A graph neural network is a graph defined by nodes and edges. Every node is a recurrent neural network, and messages between these networks can be propagated using the edges. Each node stores its own state, which is updated by means of message propagation. For each architecture, a graph neural network is generated which is homomorphic. This means we have one node for each operation in the architecture, and for each connection between operations we have an edge in the graph neural network. The hypernetwork is then conditioned on the states of the graph neural network to infer the weights of the architecture.

Discussion All of the one-shot approaches share the weights across different architectures, and rely on the hypothesis that the over-parameterized trained network can be used to rank architectures for quality. This belief is supported by Bender et al. (2018) who show a correlation between the validation error obtained for a sample of the over-parameterized network and the one obtained after training the corresponding architecture from scratch. However, they acknowledge that training the over-parameterized network is not trivial and that the practical approaches of batch normalization, dropout rate and regularization play an important role. Zhang et al. (2019) conduct a similar experiment using a separate implementation and confirm the correlation between the two variables. The correlation for the hypernetwork-based optimizers has been independently confirmed by Brock et al. (2018) and Zhang et al. (2019). Thus, it seems that we have sufficient empirical evidence that weight sharing is indeed a useful tool for the search of CNN architectures. As a result, the search only takes up insignificantly additional time than training a candidate in the search space and is consequently incredibly efficient compared to other optimization methods. Sciuto et al. (2019), challenge this widespread belief, in particular for its applicability for searching recurrent cells. They show that the ranking of architectures implied by the various methods (Pham et al., 2018; Liu et al., 2018c; Luo et al., 2018) does not correlate with the true ranking. The reason that the search methods still deliver good results is solely due to the very limited search space. They provide empirical evidence that a random search outperforms many of the previously described methods in the search for an RNN cell. We perform a similar experiment for convolutional neural networks and visualize the results in Figure 18. For the first experiment we use the publicly available source code for the method suggested by Pham et al. (2018) and apply it for searching architectures for



(a) Reinforcement Learning

(b) Uniform Sampling

Figure 18: Both one-shot methods using uniform sampling and reinforcement learning provide no correlation between the architecture ranking implied by the over-parameterized network and the actual architecture ranking on CIFAR-10.

CIFAR-10 classification. As described previously, the shared weights are learned jointly with a controller that is trained by means of reinforcement learning. Over time, this controller learns to sample better architectures such that the correspomding weights which comprise of a particular subset of the entire search space are trained more frequently. Finally, we sample some architectures with the controller and train them independently for 70 epochs from scratch. We then calculate the correlation between the validation error observed on the over-parameterized network and that after training the architecture using the Kendall Tau Rank Correlation Coefficient (Kendall, 1945). We note no correlation between these two variables (correlation of 0.12 with a p-value of 0.0003).

For the second experiment, we modify the source code to sample architectures uniformly during training, as proposed by Bender et al. (2018). This experiment is particularly interesting, as possible effects added by sophisticated sampling strategies are eliminated. After the training phase, we randomly choose some architectures and train them for 70 epochs. First, we find that the reinforcement learning controller is able to learn to sample better architectures. The true validation error increases from $6.68\% \pm 1.63\%$ to $8.98\% \pm 1.73\%$ in the uniform case. However, the Kendall Tau Rank Correlation remains low at 0.08 with a p-value of 0.0521.

At this point, we do not want to conclude that sharing weights for CNNs also does not work, as there are currently too many references in the literature that argue the contrary. However, we would like to point out that there is a certain risk and more extensive investigation is necessary.

3.5 Conclusions

In this section, we reviewed various optimization algorithms based on methods such as reinforcement learning, evolutionary algorithms, surrogate model-based optimization, and one-shot models. Table 2 gives an overview of the results obtained by the different algorithms for the classification task on CIFAR-10 benchmark dataset. Naturally, the imminent question is - which method should be prescribed? In order to answer that we need to in-

Table 2: We give an overview of the results obtained and the search time required on CIFAR-10 by the various search algorithms discussed. In addition, we list the results of various random searches and human-designed architectures.

Reference	Error (%)	Params (Millions)	GPU Days
Baker et al. (2017)	6.92	11.18	100
Zoph and Le (2017)	3.65	37.4	22,400
Cai et al. (2018a)	4.23	23.4	10
Zoph et al. (2018)	3.41	3.3	2,000
Zoph et al. (2018) + Cutout	2.65	3.3	2,000
Zhong et al. (2018)	3.54	39.8	96
Cai et al. (2018b)	2.99	5.7	200
Cai et al. (2018b) + Cutout	2.49	5.7	200
Real et al. (2017)	5.40	5.4	2,600
Xie and Yuille (2017)	5.39	N/A	17
Suganuma et al. (2017)	5.98	1.7	14.9
Liu et al. (2018b)	3.75	15.7	300
Real et al. (2019)	3.34	3.2	3,150
Elsken et al. (2018)	5.2	19.7	1
Wistuba (2018) + Cutout	3.57	5.8	0.5
Kandasamy et al. (2018)	8.69	N/A	1.7
Liu et al. (2018a)	3.41	3.2	225
Luo et al. (2018)	3.18	10.6	200
Pham et al. (2018)	3.54	4.6	0.5
Pham et al. (2018) + Cutout	2.89	4.6	0.5
Bender et al. (2018)	4.00	5.0	N/A
Casale et al. (2019) + Cutout	2.81	3.7	1
Liu et al. (2018c) + Cutout	2.76	3.3	4
Xie et al. (2019b) + Cutout	2.85	2.8	1.5
Cai et al. (2019) + Cutout	2.08	5.7	8.33
Brock et al. (2018)	4.03	16.0	3
Zhang et al. (2019)	4.30	5.1	0.4
Random (Liu et al., 2018b)	3.91	N/A	300
Random (Luo et al., 2018)	3.92	3.9	0.3
Random (Liu et al., 2018c) + Cutout	3.29	3.2	4
Random (Li and Talwalkar, 2019) + Cutout	2.85	4.3	2.7
Zagoruyko and Komodakis (2016)	3.87	36.2	-
Gastaldi (2017) (26 2x32d)	3.55	2.9	-
Gastaldi (2017) (26 2x96d)	2.86	26.2	-
Gastaldi (2017) (26 2x112d)	2.82	35.6	-
Yamada et al. (2016) + ShakeDrop	2.67	26.2	-

spect if the results can be fairly compared. In our opinion, this is not the case. The different experiments differ drastically in terms of search space, search duration and data augmentation. However, we can safely conclude that the NASNet search space is the most commonly used search space, presumably because the imposed restrictions in its definition favor the discovery of well performing architectures. Similarly, strategies that reuse parameters (through function-preserving transformations or weight sharing) are effective in reducing the search duration. However, we would like to re-emphasize the questionable benefits of weight sharing as noted in our experiments (see Figure 18).

At this point we need to criticize the lack of fair baselines. Although architecture search can be considered as a special way of optimizing hyperparameters, most of the related work is disregarded. In particular, random search which has proven to be an extremely strong baseline. In fact, Sciuto et al. (2019) show that random search finds better RNN cells than any other optimizer. Li and Talwalkar (2019) also confirm this result and additionally show that random search finds architectures that perform at least as well as the ones obtained from established optimizers for CNNs. It is worth noting that these random optimizers work on a search space that is known to sample well-performing architectures. Xie et al. (2019a) go one step further and use a graph generator to generate random graph structures that no longer adhere to the rules of established search spaces. Although these graph generators do not have any deep learning specific prior, the generated architectures perform better than those found by complex architecture optimizers. Architectures designed by humans often serve as a motivation to justify the search for neural architectures. However, state-of-the-art architectures are often not considered for comparison or not trained under equivalent experimental conditions. Thus, the effect of learning rate decay strategies, augmenting techniques (cutout (DeVries and Taylor, 2017)), regularization tricks (ScheduledDropPath (Zoph et al., 2018)), and other nuanced strategies that effect training dynamics gets overshadowed. We include the results for the state-of-the-art architectures in Table 2 and report additional results for models trained with the popular data-augmentation technique of cutout in Table 3. We note that the performance gap between human-designed and discovered architectures is smaller than what is often claimed or reported.

We recognize that in recent years many interesting and creative architecture search methods have been developed, but we argue that the source of performance gains still remains unclear. This remains a highly relevant and unanswered question for this research area.

4. Constraints and Multiple Objectives

While it is important to find networks that yield high accuracy, sometimes it is also imperative to consider other objectives, such as the number of model parameters, the number of floating point operations per second for computing model output, and device specific statistics like the latency of the model. There exist two different approaches which account for these additional objectives. In the first approach, the conditions are added as constraints to the optimization problem so as to enforce requirements like fewer parameters or faster inference time. The exact form of the constraints and the trade-off between different constraints can be adjusted as per practical requirements. In the second approach, the problem is tackled as a multi-objective function optimization problem which yields a set of proposals as a solution. This approach differs in its formalism as no preferred trade-off between different objective functions has to be specified and a solution can be selected from the set of discovered alternatives.

4.1 Constrained Optimization

For some tasks modeled with deep learning, specific constraints g_i, such as, thresholds on inference time or memory requirement are explicitly stated. The single-objective optimization problem defined in Equation (9) then turns into a constrained optimization problem as defined by

$$\max_{\boldsymbol{\alpha} \in A} \quad f\left(\boldsymbol{\alpha}\right) \tag{19a}$$

$$\text{subject to} \quad g_i\left(\boldsymbol{\alpha}\right) \leq c_i \ \forall i \in I\,. \tag{19b}$$

The optimization methods discussed in Section 3 cannot be directly adopted for this scenario as they were designed to solve an unconstrained optimization problem. However, the classical technique of penalty methods can be used to form the unconstrained optimization given by,

$$\max_{\boldsymbol{\alpha} \in A} f\left(\boldsymbol{\alpha}\right) \cdot \prod_{i \in I} \lambda\left(g_i\left(\boldsymbol{\alpha}\right), c_i\right)\,. \tag{20}$$

The function λ is a penalty function, that punishes the violation of a constraint. Once a penalty function has been selected, most optimization method discussed in Section 3 can be used to solve the problem. In the following, we discuss the various penalty functions considered in the literature along with the specific optimizer used in different approaches.

Tan et al. (2018) use

$$\lambda\left(g_i\left(\boldsymbol{\alpha}\right), c_i\right) = \left[\frac{g_i\left(\boldsymbol{\alpha}\right)}{c_i}\right]^{w_i}\,, \tag{21}$$

as a penalty function, where $\boldsymbol{w}$ is treated as a hyperparameter to suit the desired trade-off. They apply the reinforcement learning optimizer proposed by Zoph and Le (2017) to their factorized hierarchical search space as discussed in Section 2.2. Zhou et al. (2018) propose to use

$$\lambda\left(g_i\left(\boldsymbol{\alpha}\right), c_i\right) = \phi^{\max\{0,(g_i(\boldsymbol{\alpha})-c_i)/c_i\}}\,, \tag{22}$$

where ϕ is a penalization constant in the range of 0 to 1 and use a similar approach to Cai et al. (2018a) to find the final architecture. Specifically, a policy is learned that decides whether to add, remove or keep a layer as well as whether to alter its number of filters. Hsu et al. (2018) use a harder penalty function which returns 0 when the constraint is violated and 1 otherwise. They use a reinforcement learning optimizer similar to the one proposed by Zoph and Le (2017) which predicts the hyperparameters of different layers. Thus, the reward received by the controller is the accuracy for cases which do not violate any constraint and is 0 otherwise. The authors evaluate their optimizer to select the layer-wise hyperparameters (number of filters, kernel size) of an AlexNet (Krizhevsky et al., 2012) and the cell-wise hyperparameters (stage, growth rate) of a CondenseNet (Huang et al., 2018a). We suggest to refer to Section 3.1 for more details on the optimization methods.

4.2 Multi-Objective Optimization

Another approach to handle multiple fronts lies in formalism of multi-objective optimization problem as defined as

$$\max_{\boldsymbol{\alpha} \in A} f_1\left(\boldsymbol{\alpha}\right), f_2\left(\boldsymbol{\alpha}\right), \ldots, f_n\left(\boldsymbol{\alpha}\right) . \tag{23}$$

However, often there is no single optimal solution that minimizes all these functions. As some of the objectives can be conflicting, the best solution with respect to one objective might not yield the best solution with respect to another objective. Therefore, the task is to find a set of solutions which are *Pareto optimal.* A solution is Pareto optimal if none of the objectives can be improved without worsening at least one other objective. This means that for a Pareto optimal solution $\boldsymbol{\alpha}$, no other solution $\boldsymbol{\alpha}' \in A$ exists such that $f_i\left(\boldsymbol{\alpha}\right) \geq f_i\left(\boldsymbol{\alpha}'\right)$ with $f_j\left(\boldsymbol{\alpha}\right) > f_j\left(\boldsymbol{\alpha}'\right)$ for some j. Otherwise, we say $\boldsymbol{\alpha}'$ dominates $\boldsymbol{\alpha}$, $\boldsymbol{\alpha}' \prec \boldsymbol{\alpha}$. Finally, it is up to the user to select a model from the set of Pareto optimal solutions, the Pareto front, maybe depending on different deployment scenarios.

Decomposition Methods One way to solve this problem is the decomposition approach. A parameterized aggregation function h is used to transform the multi-objective optimization problem into a single-objective optimization problem

$$\max_{\boldsymbol{\alpha} \in A} h\left(\left(f_1\left(\boldsymbol{\alpha}\right), f_2\left(\boldsymbol{\alpha}\right), \ldots, f_n\left(\boldsymbol{\alpha}\right)\right), \boldsymbol{w}\right) . \tag{24}$$

Examples for h are the weighted sum, weighted exponential sum, weighted min-max or weighted product (Santiago Pineda et al., 2014). However, solving this problem for a fixed setting of the weights $\boldsymbol{w}$ will in most cases not find all Pareto optimal solutions. Therefore, the problem is solved for different $\boldsymbol{w}$. In neural architecture search this requires multiple optimization runs for several different weight vectors which is prohibitive. Therefore, a common approach is to select an aggregation function and fix the weight vector according to domain knowledge and the desired trade-off between objectives. Consequently, the multi-objective becomes a single-objective optimization problem which can be solved by any method discussed in Section 3. Hsu et al. (2018) propose to use the weighted sum as the aggregation function and use a reinforcement learning approach (Zoph and Le, 2017) to solve this problem. Unfortunately, not every objective function is differentiable which is a requirement for some of the optimization methods discussed in Section 3.4. Cai et al. (2019) demonstrate one way to overcome this obstacle. They propose to replace non-differentiable objective functions with differentiable surrogate models. Specifically, they use a surrogate model to predict the latency of an operation.

NSGA-II An alternative approach is to estimate architectures which are not dominated by the current solution set and evaluate their performance. NSGA-II (Deb et al., 2002) is an elitist evolutionary algorithm that is utilized predominantly. The initial population is selected at random. Candidate solutions are sorted into different lists, called fronts. All the non-dominated solutions belong to the first front. The solutions in the i^{th} front are only dominated by all the solutions in the $1, \ldots, i-1$ fronts. Solutions within a front are sorted according to their crowding distance which is a measure for the density of solutions within this solution's region. It is computed by the sum of all the neighborhood distances across all the objectives and ensures that the algorithm explores diverse solutions in the search space. This yields a ranking of all solutions where the solution in front 1 with lowest density is the best one. Now, the top k solutions are selected as parents, mutated, recombined and evaluated.

Kim et al. (2017) and Lu et al. (2018) both employ NSGA-II in order to tackle the problem. The early work of Kim et al. (2017) encodes the genotype by means of layer type and number of outputs leading to only sequential networks. In their experiments Lu et al. (2018) report results on both NASNet search space and the genotype representation by Xie and Yuille (2017) (see Section 2 for details) and therefore are not limited to sequential networks. They use the vanilla NSGA-II for modeling the evolutionary process. However the sampling step in their approach comprises of two phases - exploration and exploitation. For the exploration phase, the offspring is generated by applying mutations and cross-over. In the exploitation phase new samples are obtained from a Bayesian network that models the distribution of previously trained architectures.

Elsken et al. (2019) Elsken et al. (2019) propose another evolutionary algorithm which shares many common aspects with their earlier work (Elsken et al., 2018) (see Section 3.2). No cross-over operations are used and mutations are function-preserving. They extend the pool of mutations and include new ones that shrink an architecture by means of layer removal or filter reduction. Similar to NSGA-II, their algorithm tries to achieve diverse solutions. They take advantage of the fact that the objective functions can be divided into *cheap-to-evaluate objective functions* (e.g. number of parameters, inference time etc.) and *expensive-to-evaluate objective functions* (e.g. validation accuracy). A kernel density estimator is used to model the density of candidates with respect to the cheap-to-evaluate objectives in order to select candidates from low-density regions. Finally, the expensive-to-evaluate objectives for the candidates are evaluated and the Pareto front is updated.

Smithson et al. (2016) Smithson et al. (2016) propose a surrogate model-based approach to search for a sequential architecture. The search algorithm starts with a random parameter setting. New candidates are sampled from a Gaussian distribution around previously evaluated architectures. A surrogate model (artificial neural network) is used to predict the values for the expensive-to-evaluate objective functions, cheap-to-evaluate objectives are evaluated exactly. Based on both the predictions and evaluations, it is determined whether this solution is dominated by an existing solution which will in turn determine the probability the candidate is accepted. Every accepted candidate will be evaluated and thereafter the surrogate model is updated.

Dong et al. (2018) Dong et al. (2018) is another surrogate model-based approach which extends the work by Liu et al. (2018a) (see Section 3.3) to solve multi-objective optimization problems. The only difference of the optimization method is how models are selected. Instead of only considering model accuracy, they also consider all other objectives. Feasible candidates are those which do not violate any constraints and are not dominated (according to the prediction of the surrogate model) by any known solution.

5. Other Related Methods

Methodologies originally developed for architecture search have been extended to automate other aspects of deep learning. For instance, Bello et al. (2017) encode a parameter opti-

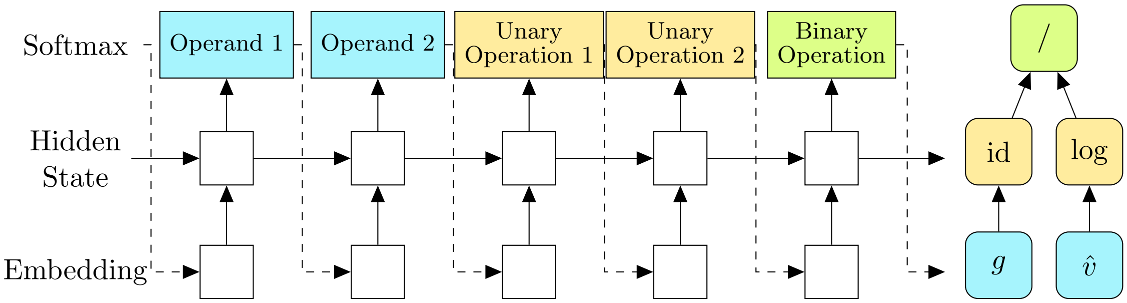


Figure 19: Controller output for one group. Each group consists of two operands and the unary operations applied to it as well as the binary operation which combines the outcome of the unary operations.

mization method used in training of a deep learning model by means of a graph structure that defines the update rules akin to classical gradient-based optimization. An update is defined by several blocks as shown in Figure 19. A block consists of two operands, one unary function for each of the operands and a binary function. The unary functions are applied on the operands and the binary function is applied to combine the respective outcomes. The output of this block becomes a possible operand itself and can be used by other blocks. This structure is stacked up to a depth of three. Possible operands include different orders of the gradient (g, g^2, g^3), its bias-corrected running estimates, differently scaled versions of the parameter, random noise, constants, and the sign of the gradient. Additionally the classical updates of Adam and RMSProp optimizer are also encapsulated as operands. Examples for unary functions are the identity function, clipping functions, the sign function, functions that modify the gradient to zero with a certain probability, as well as logarithmic and exponential function. Binary functions include addition, subtraction, multiplication, and division. A reinforcement controller, similar to the one proposed by Zoph and Le (2017) (see Section 3.1), is applied to search for an optimal structure. The main difference in the work of Bello et al. (2017) is that they use Trust Region Policy Optimization (Schulman et al., 2015) instead of REINFORCE for updating the controller parameters. A deep-learning optimizer sampled from the controller is used to train a simple two-layer convolutional neural network for only five epochs and the validation accuracy is used as the reward.

Ramachandran et al. (2018) uses the aforementioned setup and search space structure to search for activation functions. They learn the controller with Proximal Policy Optimization (Schulman et al., 2017). As they seek to search for activation functions, there setup comprises of only one operand which is the input of the activation function and a different set of unary and binary operations. Examples for unary operations are powers of different degree, various sigmoid functions and the sine function. The set of possible binary functions on the input (x_1, x_2) is extended with additional functions like $x_1 \cdot \sigma(x_2)$, maximum and minimum, and a weighted average. The reward is estimated by training a ResNet-20 with the sampled activation function on CIFAR-10 for 10,000 steps. The authors report the discovery of the following activation function

$$\mathrm{swish}(x) = x \cdot \sigma(x) \, , \tag{25}$$

Table 3: Classification error of various architectures with different augmentation strategies. Whenever two numbers are reported, the left one are the results obtained by us then the right one the results reported by Cubuk et al. (2018a). Cubuk et al. (2018a) do not report results for mixup.

Model	Baseline	Cutout	Mixup	Cutout +Mixup	Auto-Augment	Auto-Augment +Mixup
Wide-ResNet (28-10)	3.93/3.87	2.89/3.08	3.24	2.95	2.71/2.68	2.60
Shake-Shake (26 2x32d)	3.73/3.55	3.03/3.02	3.13	3.10	2.59/2.47	2.85
Shake-Shake (26 2x96d)	3.15/2.86	2.50/2.56	2.37	2.15	2.16/1.99	1.84
Shake-Shake (26 2x112d)	3.13/2.82	2.55/2.57	2.37	2.12	2.09/1.89	1.83
PyramidNet +ShakeDrop	2.77/2.67	2.09/2.31	1.90	1.51	1.62/1.48	1.33

where σ is the logistic function. In their experiments, this particular activation function turned out to be more effective than other commonly used activation functions and has been used by many other authors since. However, it is interesting to note that the discovered function closely resembles the previously proposed swish activation in an earlier work by the authors.

Cubuk et al. (2018a) define a similar controller and train it with Proximal Policy Optimization to optimize data augmentation policies for image classification. The controller predicts five sub-policies, each consists of two augmentation operations and always uses horizontal flipping, random crops and a cutout operation. Examples for augmentation operations are rotation, sheering, control the brightness, contrast and sharpness of the image. For every image, a random sub-policy is selected and applied. The validation accuracy is used as the reward signal. Finally, the best sub-policies of five policies are concatenated to the final augmentation policy which consists of 25 sub-policies.

Experiment We conduct an experiment to better understand the impact of established augmentation schemes and contrast it with the found policies. As part of this we reconducted the experiments by Cubuk et al. (2018a) using their publicly available code. A Wide-ResNet (Zagoruyko and Komodakis, 2016), three versions of Shake-Shake (Gastaldi, 2017) and a PyramidNet (Yamada et al., 2016) with ShakeDrop (Yamada et al., 2018) are trained with different augmentation strategies as detailed in the following. *Baseline* is horizontal flipping and random crops and *cutout* additionally uses cutout (DeVries and Taylor, 2017). *AutoAugment* extends the cutout policy by augmentation sub-policies found during the search. We extend this setup further by considering mixup (Zhang et al., 2018) as another augmentation technique in order to see whether the AutoAugment policy can improve over standard state-of-the-art augmentation techniques. As we see in Table 3, the combination of cutout with mixup in many cases is better than using just one of them. Since AutoAugment is cutout plus other augmentation techniques, perhaps it is not surprising to notice a significant lift over cutout. We consider the combination of cutout and mixup as a useful baseline which it is not able to outperform. However, adding mixup to AutoAugment does not hurt the performance in most cases as expected. In cases where the combination of cutout and mixup performed better than AutoAugment, this addition provides a significant improvement resulting in the best scores. We are able to achieve an error of 1.33% on CIFAR-10 classification task. To the best of our knowledge, the only work claiming a lower error on this task is the work by Huang et al. (2018b) which claims 1% error. However, they use a pretrained model on ImageNet which has 557 million parameters, while the PyramidNet used in our experiments has “only” 26.2 million parameters can be trained on a single GPU. More importantly, we do not use any pretrained weights and train the model from scratch.

6. Outlook and Future Applications of Neural Architecture Search Methods

Currently, the approaches in neural architecture search focus on CNNs for solving the task of object recognition and RNN cells for language modeling. However, other interesting areas have already surfaced some of which have been discussed in sections 4 and 5. For example, the search for architectures under constraints and with multiple objectives has garnered significant attention in the past year (Section 4). The alignment of architecture optimization algorithms with other graph structures such as heuristic optimization algorithms and activation functions or the optimization of data augmentation are exciting developments (Section 5). Recently there has also been promising development along other dimensions that address more complex tasks with image data such as object detection and segmentation (Zoph et al., 2018; Tan et al., 2018; Liu et al., 2019; Weng et al., 2019), as well as works that address safety-critical issues such as the discovery of architectures that are robust against adversarial attacks (Cubuk et al., 2018b; Sinn et al., 2019). Furthermore, the existing techniques are extended to apply the architecture search for other types of networks. For instance, there are first attempts to automatically optimize the architecture of autoencoders (Suganuma et al., 2018), transformers (So et al., 2019) and graph CNNs (Gao et al., 2019). In general, these techniques are increasingly being used to automate all components of the data science work flow. In addition to the automation of data augmentation discussed earlier, there is also initial work that proposes to automate the compression of networks (He et al., 2018). Another interesting idea that has been used in classical machine learning automation for decades is the idea of using knowledge across different datasets to speed up the optimization processes. While this niche has been tackled by only few works for neural architecture search (Wong et al., 2018), we expect to see more activity in this area in the coming years. The big remaining challenge is the joint optimization of all configuration parameters of the deep learning work flow. So far, the individual components have been treated independently with some components still relying on manual configuration. However, for true automation, it is inevitable that all sensitive parameters must be learned or searched for. This includes data augmentation, the various search space hyperparameters, the choice of optimizer, the actual architecture choice and possibly even the final model compression.

One important question which we believe is often overlooked pertains to the credibility of neural architecture search as a meaningful field of research. Has it progressed the field of machine learning or provided us new insights in deep learning? We would like to answer this questions with a "maybe". It is crucial to emphasize the role of search space designs in the performance of obtained architectures, some of which are heavily inspired by the design of existing architectures. Many architectures originate from the NASNet search space, which were found by various search algorithms. Therefore, a legitimate question is to ask whether the search algorithms have really discovered new, innovative architectures or whether they owe their success to the design and properties of the NASNet search space. Li and Talwalkar (2019), we as well as the inventors of the NASNet search space, have already partially answered this question by showing that even a random search yields very good, if not better, results. Furthermore, these architectures ultimately do not seem to be significantly better than those developed by humans. The human innovations that contribute to the invention of new architectures are difficult to discover given the restrictions of search spaces. This brings us to the question of what to expect from the future developments of architecture search algorithms. If we really seek to find new architectures with innovative elements that advance research in general, then perhaps we have to let go of the restrictive search spaces like NASNet. While a general automation of deep learning might answer most of these concerns, perhaps for now we need to ask ourselves why an architecture discovered by a search algorithm is better than the choice of an existing architecture in combination with hyperparameter optimization. The answer can not be that each dataset needs its own architecture. Many architecture search papers show that found architectures can be easily transferred to other datasets, even other tasks, which is a well known property of deep learning architectures. So why then should a practitioner commit to the use of a search algorithm for finding a new architecture, in particular since it does not free her from the optimization of the search space hyperparameters?

The general automation of deep learning is still in its infancy, and many of the aforementioned concerns remain unanswered for the time being. However, this remains an exciting domain and the future works will certainly highlight its practical relevance and usefulness.

References

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL `https://openreview.net/forum?id=S1c2cvqee`.

Friedrich L Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 459–468, 2017. URL `http://proceedings.mlr.press/v70/bello17a.html`.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 550–559, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/bender18a.html.

Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002. doi: 10.1023/A:1015059928466. URL https://doi.org/10.1023/A:1015059928466.

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. URL https://openreview.net/forum?id=rydeCEhs-.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *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, 2018a. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16755.

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 677–686, 2018b. URL http://proceedings.mlr.press/v80/cai18a.html.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proceedings of the International Conference on Learning Representations, ICLR 2019, New Orleans, Louisiana, USA*, 2019. URL https://openreview.net/forum?id=HylVB3AqYm.

Francesco Paolo Casale, Jonathan Gordon, and Nicolo Fusi. Probabilistic neural architecture search. *CoRR*, abs/1902.05116, 2019. URL http://arxiv.org/abs/1902.05116.

Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. 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. URL http://arxiv.org/abs/1511.05641.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *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.

URL http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations.

Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501, 2018a. URL http://arxiv.org/abs/1805.09501.

Ekin Dogus Cubuk, Barret Zoph, Samuel S. Schoenholz, and Quoc V. Le. Intriguing properties of adversarial examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018b. URL https://openreview.net/forum?id=Skz1zaRLz.

Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6 (2):182–197, 2002. doi: 10.1109/4235.996017. URL https://doi.org/10.1109/4235.996017.

Terrance DeVries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017. URL http://arxiv.org/abs/1708.04552.

Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dppnet: Device-aware progressive search for pareto-optimal neural architectures. 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 XI*, volume 11215 of *Lecture Notes in Computer Science*, pages 540–555. Springer, 2018. doi: 10.1007/978-3-030-01252-6_32. URL https://doi.org/10.1007/978-3-030-01252-6_32.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018. URL https://openreview.net/forum?id=H1hymrkDf.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *Proceedings of the International Conference on Learning Representations, ICLR 2019, New Orleans, Louisiana, USA*, 2019.

Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008. URL https://doi.org/10.1007/s12065-007-0002-4.

Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. Graphnas: Graph neural architecture search with reinforcement learning. *CoRR*, abs/1904.09981, 2019. URL http://arxiv.org/abs/1904.09981.

Xavier Gastaldi. Shake-shake regularization. *CoRR*, abs/1705.07485, 2017. URL http://arxiv.org/abs/1705.07485.

David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Proceedings of the First Workshop on Foundations of Genetic Algorithms. Bloomington Campus, Indiana, USA, July 15-18 1990.*, pages 69–93. Morgan Kaufmann, 1990.

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

David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL `https://openreview.net/forum?id=rkpACe1lx`.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: automl for model compression and acceleration on mobile devices. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VII*, pages 815–832, 2018. doi: 10.1007/978-3-030-01234-2_48. URL `https://doi.org/10.1007/978-3-030-01234-2_48`.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`.

Chi-Hung Hsu, Shu-Huan Chang, Da-Cheng Juan, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Shih-Chieh Chang. MONAS: multi-objective neural architecture search using reinforcement learning. *CoRR*, abs/1806.10332, 2018.

Hanzhang Hu, John Langford, Rich Caruana, Eric Horvitz, and Debadeepta Dey. Macro neural architecture search revisited. In *Workshop on Meta-Learning at NeurIPS 2018, MetaLearn 2018, 3-8 December 2018, Montréal, Canada.*, 2018. URL `http://metalearning.ml/2018/papers/metalearn2018_paper16.pdf`.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.243. URL `https://doi.org/10.1109/CVPR.2017.243`.

Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2752–2761. IEEE Computer Society, 2018a. doi: 10.1109/CVPR.2018.00291. URL `http://openaccess.thecvf.com/content_cvpr_2018/html/Huang_CondenseNet_An_Efficient_CVPR_2018_paper.html`.

Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018b. URL `http://arxiv.org/abs/1811.06965`.

Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. Neural architecture search with bayesian optimisation and optimal transport. 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 2020–2029, 2018. URL `http://papers.nips.cc/paper/7472-neural-architecture-search-with-bayesian-optimisation-and-optimal-transport`.

M. G. Kendall. The treatment of ties in ranking problems. *Biometrika*, 33(3):239–251, 11 1945. ISSN 0006-3444. doi: 10.1093/biomet/33.3.239. URL `https://dx.doi.org/10.1093/biomet/33.3.239`.

Ye-Hoon Kim, Bhargava Reddy, Sojung Yun, and Chanwon Seo. Nemo : Neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy. In *AutoML Workshop at ICML 2017*, 2017.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, Canadian Institute for Advanced Research, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *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. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks`.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638, 2019. URL `http://arxiv.org/abs/1902.07638`.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018a.

Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L. Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *CoRR*, abs/1901.02985, 2019. URL `http://arxiv.org/abs/1901.02985`.

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*, 2018b. URL `https://openreview.net/forum?id=BJQRKzbA-`.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018c. URL `http://arxiv.org/abs/1806.09055`.

Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh D. Dhebar, Kalyanmoy Deb, Erik D. Goodman, and Wolfgang Banzhaf. NSGA-NET: A multi-objective genetic algorithm for neural architecture search. *CoRR*, abs/1810.03522, 2018. URL `http://arxiv.org/abs/1810.03522`.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. 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 7827–7838, 2018. URL `http://papers.nips.cc/paper/8007-neural-architecture-optimization`.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL `https://openreview.net/forum?id=S1jE5L5gl`.

Jonas Mockus, Vytautas Tiešis, and Antanas Žilinskas. The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 2:117–129, 1978.

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/pham18a.html`.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018. URL `https://openreview.net/forum?id=Hkuq2EkPf`.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006. ISBN 026218253X. URL `http://www.worldcat.org/oclc/61285753`.

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*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL `http://proceedings.mlr.press/v70/real17a.html`.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Aging evolution for image classifier architecture search. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, (AAAI-19), Honolulu, Hawaii, USA*, 2019.

Alejandro Santiago Pineda, Hctor Joaqun Fraire Huacuja, Bernabé Dorronsoro, Johnatan E. Pecero, Claudia Gómez Santillán, Juan Javier González Barbosa, and Jos'e Carlos Soto Monterrubio. A survey of decomposition methods for multi-objective optimization. In *Recent Advances on Hybrid Approaches for Designing Intelligent Systems*, pages 453–465. Springer, 2014. doi: 10.1007/978-3-319-05170-3_31. URL `https://doi.org/10.1007/978-3-319-05170-3_31`.

Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4053–4061, 2016. URL `http://papers.nips.cc/paper/6304-convolutional-neural-fabrics`.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605. URL `https://doi.org/10.1109/TNN.2008.2005605`.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1889–1897, 2015. URL `http://jmlr.org/proceedings/papers/v37/schulman15.html`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Processing*, 45(11):2673–2681, 1997. doi: 10.1109/78.650093. URL `https://doi.org/10.1109/78.650093`.

Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *CoRR*, abs/1902.08142, 2019. URL `http://arxiv.org/abs/1902.08142`.

Mathieu Sinn, Martin Wistuba, Beat Buesser, Maria-Irina Nicolae, and Minh Tran. Evolutionary search for adversarially robust neural networks. *Safe Machine Learning Workshop at ICLR 2019, New Orleans, Louisiana, USA*, 2019.

Sean C. Smithson, Guang Yang, Warren J. Gross, and Brett H. Meyer. Neural networks designing neural networks: multi-objective hyper-parameter optimization. In *ICCAD*, page 104. ACM, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. 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 2960–2968, 2012. URL `http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms`.

David R. So, Chen Liang, and Quoc V. Le. The evolved transformer. *CoRR*, abs/1901.11117, 2019. URL http://arxiv.org/abs/1901.11117.

Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 497–504, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4920-8. doi: 10.1145/3071178.3071229. URL http://doi.acm.org/10.1145/3071178.3071229.

Masanori Suganuma, Mete Özay, and Takayuki Okatani. Exploiting the potential of standard convolutional autoencoders for image restoration by evolutionary search. *CoRR*, abs/1803.00370, 2018. URL http://arxiv.org/abs/1803.00370.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998. ISBN 0262193981. URL http://www.worldcat.org/oclc/37293240.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018. URL http://arxiv.org/abs/1807.11626.

Cdric Villani. *Optimal Transport: Old and New.* Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008. ISBN 9783540710509.

Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK, May 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Yu Weng, Tianbao Zhou, Yujie Li, and Xiaoyu Qiu. Nas-unet: Neural architecture search for medical image segmentation. *IEEE Access*, 7:44247–44257, 2019. doi: 10.1109/ACCESS.2019.2908991. URL https://doi.org/10.1109/ACCESS.2019.2908991.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Martin Wistuba. Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations. In *ECML/PKDD (2)*, volume 11052 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2018. URL http://www.ecmlpkdd2018.org/wp-content/uploads/2018/09/108.pdf.

Catherine Wong, Neil Houlsby, Yifeng Lu, and Andrea Gesmundo. Transfer learning with neural automl. 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 8366–8375, 2018. URL http://papers.nips.cc/paper/8056-transfer-learning-with-neural-automl.

Lingxi Xie and Alan L. Yuille. Genetic CNN. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 1388–1397. IEEE

Computer Society, 2017. doi: 10.1109/ICCV.2017.154. URL https://doi.org/10.1109/ICCV.2017.154.

Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *CoRR*, abs/1904.01569, 2019a. URL http://arxiv.org/abs/1904.01569.

Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *Proceedings of the International Conference on Learning Representations, ICLR 2019, New Orleans, Louisiana, USA*, 2019b.

Yoshihiro Yamada, Masakazu Iwamura, and Koichi Kise. Deep pyramidal residual networks with separated stochastic depth. *CoRR*, abs/1612.01230, 2016. URL http://arxiv.org/abs/1612.01230.

Yoshihiro Yamada, Masakazu Iwamura, and Koichi Kise. Shakedrop regularization. *CoRR*, abs/1802.02375, 2018. URL http://arxiv.org/abs/1802.02375.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016*, 2016. URL http://www.bmva.org/bmvc/2016/papers/paper087/index.html.

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. In *Proceedings of the International Conference on Learning Representations, ICLR 2019, New Orleans, Louisiana, USA*, 2019.

Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. URL https://openreview.net/forum?id=r1Ddp1-Rb.

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2423–2432, 2018. doi: 10.1109/CVPR.2018.00257. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Zhong_Practical_Block-Wise_Neural_CVPR_2018_paper.html.

Yanqi Zhou, Siavash Ebrahimi, Sercan Ömer Arik, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *CoRR*, abs/1806.07912, 2018. URL http://arxiv.org/abs/1806.07912.

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. URL https://openreview.net/forum?id=r1Ue8Hcxg.

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. doi: 10.1109/CVPR.2018.00907. URL `http://openaccess.thecvf.com/content_cvpr_2018/html/Zoph_Learning_Transferable_Architectures_CVPR_2018_paper.html`.