# EFFICIENT MULTI-OBJECTIVE NEURAL ARCHITECTURE SEARCH VIA LAMARCKIAN EVOLUTION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Neural Architecture search aims at automatically finding neural architectures that are competitive with architectures designed by human experts. While recent approaches have achieved state-of-the-art predictive performance for image recognition, they are problematic under resource constraints for two reasons: (1) the neural architectures found are solely optimized for high predictive performance, without penalizing excessive resource consumption; (2) most architecture search methods require vast computational resources. We address the first shortcoming by proposing `LEMONADE`, an evolutionary algorithm for multi-objective architecture search that allows approximating the entire Pareto-front of architectures under multiple objectives, such as predictive performance and number of parameters, in a single run of the method. We address the second shortcoming by proposing a Lamarckian inheritance mechanism for `LEMONADE` which generates children networks that are warmstarted with the predictive performance of their trained parents. This is accomplished by using approximate network morphism operators for generating children. The combination of these two contributions allows finding models that are on par or even outperform both hand-crafted as well as automatically-designed networks.

## 1 INTRODUCTION

Deep learning has enabled remarkable progress on a variety of perceptual tasks, such as image recognition (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012), and machine translation (Bahdanau et al., 2015). One crucial aspect for this progress are novel *neural architectures* (Szegedy et al., 2016; He et al., 2016b; Huang et al., 2017b). Currently employed architectures have mostly been developed manually by human experts, which is a time-consuming and error-prone process. Because of this, there is growing interest in automatic *architecture search* methods (Baker et al., 2017a; Zoph & Le, 2017; Real et al., 2017). Some of the architectures found in an automated way have already achieved similar predictive performance as the best manually-designed ones; however, current algorithms for finding these architectures require enormous computational resources often in the range of thousands of GPU days.

Prior work on architecture search has typically framed the problem as a single-objective optimization problem. However, most applications of deep learning do not only require high predictive performance on unseen data but also low *resource-consumption* in terms of, e.g., inference time, model size or energy. Moreover, there is typically an implicit trade-off between predictive performance and consumption of resources. Recently, several architectures have been manually designed that aim at reducing resource-consumption while retaining high predictive performance (Iandola et al., 2016; Howard et al., 2017; Sandler et al., 2018). Automatically-found neural architectures have also been down-scaled to reduce resource consumption (Zoph et al., 2018). However, very little previous work has taken the trade-off between resource-consumption and predictive performance into account during automatic architecture search.

In this work, we make the following two main contributions:

1. To overcome the need for thousands of GPU days (Zoph & Le, 2017; Zoph et al., 2018; Real et al., 2018), we make use of operators acting on the space of neural network architectures that preserve the function a network represents, dubbed *network morphisms* (Chen et al., 2015; Wei et al., 2016), obviating training from scratch and thereby substantially reducing the required training time per network. This mechanism can be interpreted as Lamarckian inheritance in the context of evolutionary algorithms, where Lamarckism refers to a mechanism which allows passing

skills acquired during an individual's lifetime (e.g., by means of learning), on to children by means of inheritance. Since network morphisms are limited to solely increasing a network's size (and therefore likely also resource consumption), we introduce *approximate network morphisms* (Section 3) to also allow shrinking networks, which is essential in the context of multi-objective search. The proposed Lamarckian inheritance mechanism could in principle be combined with any evolutionary algorithm for architecture search, or any other method using (a combination of) localized changes in architecture space.

2. We propose a multi-objective evolutionary algorithm for neural architecture search (NAS), dubbed LEMONADE, Section 4, which is suited for the joint optimization of several objectives, such as predictive performance, inference time, or number of parameters. LEMONADE maintains a population of networks on an approximation of the Pareto front of the multiple objectives. In contrast to generic multi-objective algorithms, LEMONADE exploits that evaluating certain objectives (such as an architecture's number of parameters) is cheap while evaluating the predictive performance on validation data is expensive (since it requires training the model first). Thus, LEMONADE handles its various objectives differently: it first selects a subset of architectures, assigning higher probability to architectures that would fill gaps on the Pareto front for the "cheap" objectives; then, it trains and evaluates only this subset, further reducing the computational resource requirements during architecture search. In contrast to other multi-objective architecture search methods, LEMONADE (i) does not require to define a trade-off between performance and other objectives a-priori (e.g., by weighting objectives when using scalarization methods) but rather returns a set of architectures, which allows the user to select a suitable model a-posteriori; (ii) LEMONADE does not require to be initialized with well performing architectures; it can be initialized with trivial architectures and hence requires less prior knowledge. Also, LEMONADE can handle various search spaces, including complex topologies with multiple branches and skip connections.

We evaluate LEMONADE on two different search spaces for image classification: (i) non-modularized architectures and (ii) cells that are used as repeatable building blocks within an architecture (Zoph et al., 2018; Zhong et al., 2018) and also allow transfer to other datasets. In both cases, LEMONADE returns a population of CNNs covering architectures with 10 000 to 10 000 000 parameters.

Within only one week on eight GPUs, LEMONADE discovers architectures that are competitive in terms of predictive performance and resource consumption with hand-designed networks, such as MobileNet V1, V2 (Howard et al., 2017; Sandler et al., 2018), as well as architectures that were automatically designed using 40x greater resources (Zoph et al., 2018) and other multi-objective methods (Dong et al., 2018).

## 2 Background and Related Work

**Multi-objective Optimization** Multi-objective optimization (Miettinen, 1999) deals with problems that have multiple, complementary objective functions $f_1, \dots, f_n$. Let $\mathcal{N}$ be the space of feasible solutions $N$ (in our case the space of feasible neural architectures). In general, multi-objective optimization deals with finding $N^* \in \mathcal{N}$ that minimizes the objectives $f_1, \dots, f_n$. However, typically there is no single $N^*$ that minimizes all objectives at the same time. In contrast, there are multiple *Pareto-optimal* solutions that are optimal in the sense that one cannot reduce any $f_i$ without increasing at least one $f_j$. More formally, a solution $N^{(1)}$ Pareto-dominates another solution $N^{(2)}$ if $\forall i \in 1, \dots, n : f_i(N^{(1)}) \leq f_i(N^{(2)})$ and $\exists\, j \in 1, \dots, n : f_j(N^{(1)}) < f_j(N^{(2)})$. The Pareto-optimal solutions $N^*$ are exactly those solutions that are not dominated by any other $N \in \mathcal{N}$. The set of Pareto optimal $N^*$ is the so-called *Pareto-front*.

**Neural Architecture Search** Recently, NAS was framed as a *reinforcement learning* (RL) problem, where the reward of the RL agent is based on the validation performance of the trained architecture (Baker et al., 2017a; Zoph & Le, 2017; Zhong et al., 2018; Pham et al., 2018). Zoph & Le (2017) use a recurrent neural network to generate a string representing the neural architecture. In a follow-up work, Zoph et al. (2018) search for *cells*, which are repeated according to a fixed meta architecture to generate the eventual architecture. Defining the architecture based on a cell simplifies the search space.

An alternative to using RL are *neuro-evolutionary* approaches that use genetic algorithms for optimizing the neural architecture (Stanley & Miikkulainen, 2002; Liu et al., 2018a; Real et al., 2018; Miikkulainen et al., 2017; Xie & Yuille, 2017). In contrast to these works, our proposed method is applicable for multi-objective optimization and employs Lamarckian inheritance, i.e, learned parameters are passed on to a network's offspring.

Unfortunately, most of the aforementioned approaches require vast computational resources since they need to train and validate thousands of neural architectures; e.g., Zoph & Le (2017) trained over 10.000 neural architectures, requiring thousands of GPU days. One way of speeding up evaluation is to *predict performance* of a (partially) trained model (Domhan et al., 2015; Baker et al., 2017b; Klein et al., 2017; Liu et al., 2017). Works on performance prediction are complementary to our work and could be incorporated in the future.

One-Shot Architecture Search is another promising approach for speeding up performance estimation, which treats all architectures as different subgraphs of a supergraph (the one-shot model) and shares weights between architectures Saxena & Verbeek (2016); Brock et al. (2017); Pham et al. (2018); Liu et al. (2018b); Bender et al. (2018). Only the weights of a single one-shot model need to be trained, and architectures (which are just subgraphs of the one-shot model) can then be evaluated without any separate training. However, a general limitation of one-shot NAS is that the supergraph defined a-priori restricts the search space to its subgraphs. Moreover, approaches which require that the entire supergraph resides in GPU memory during architecture search will be restricted to relatively small supergraphs. `LEMONADE` does not suffer from any of these disadvantages; it can handle arbitrary large, unconstrained search spaces while still being efficient.

Elsken et al. (2017); Cai et al. (2018a) proposed to employ the concept of *network morphisms* (see Section A.1). The basic idea is to initialize weights of newly generated neural architectures based on weights of similar, already trained architectures so that they have the same accuracy. This pretrained initialization allows reducing the large cost of training all architectures from scratch. Our work extends this approach by introducing *approximate network morphisms*, making the use of such operators suitable for multi-objective optimization.

We refer to Elsken et al. (2018) for a comprehensive survey on Neural Architecture Search.

**Multi-objective Neural Architecture Search** Very recently, there has also been some work on multi-objective neural architecture search (Kim et al., 2017; Dong et al., 2018; Tan et al., 2018) with the goal of not solely optimizing the accuracy on a given task but also considering resource consumption. Kim et al. (2017) parameterize an architecture by a fixed-length vector describing, which limits the architecture search space drastically. Dong et al. (2018) (which is parallel independent work to ours) extend PNAS (Liu et al., 2017) by considering multiple objective during the model selection step. However, they employ CondenseNet (Huang et al., 2017a) as a base network and solely optimize building blocks within the network which makes the search less interesting as (i) the base network is by default already well performing and (ii) the search space is again limited. Tan et al. (2018) use a weighted product method (Deb & Kalyanmoy, 2001) to obtain a scalarized objective. However, this scalarization comes with the drawback of weighting the objectives *a-priori*, which might not be suitable for certain applications. In contrast to all mentioned work, `LEMONADE` (i) does not require a complex meta architecture but rather can start from trivial initial networks, (ii) can handle arbitrary search spaces, (iii) does not require to define hard constraints or weights on objectives a-priori.

## 3 Approximate Network Morphisms

An operator acting on some space of neural networks maps a neural network (and its parameters) to another neural network with a (possibly) different neural architecture (e.g., more or less layers). Formally, let $\mathcal{N}(\mathcal{X})$ denote a space of neural networks, where each element $N \in \mathcal{N}(\mathcal{X})$ is a mapping from $\mathcal{X} \subset \mathbb{R}^n$ to some other space, e.g., mapping images to labels. A network operator $T : \mathcal{N}(\mathcal{X}) \times \mathbb{R}^k \to \mathcal{N}(\mathcal{X}) \times \mathbb{R}^j$ maps a neural network $N^w \in \mathcal{N}(\mathcal{X})$ with parameters $w \in \mathbb{R}^k$ to another neural network $(TN)^{\tilde{w}} \in \mathcal{N}(\mathcal{X}), \tilde{w} \in \mathbb{R}^j$. One specific class of network operators are *network morphisms* (Wei et al., 2016; Elsken et al., 2017; Cai et al., 2018a). Network morphisms by definition have the property that $N^w$ and $(TN)^{\tilde{w}}$ represent the same function, meaning $N^w(x) = (TN)^{\tilde{w}}(x)$ for every $x \in \mathcal{X}$. This can be achieved by properly choosing $\tilde{w}$. Operations such as

inserting a layer, increasing the number of filters in a convolution or adding a skip connection can be framed as network morphisms. We refer to A.1 for a comprehensive recap.

Unfortunately, network morphism operators have one thing in common: they increase the capacity of the network[1]. This may be a reasonable property if one solely aims at finding a neural architecture with maximal accuracy, but not if one also aims at neural architectures with low resource requirements. Also, decisions once made can not be reverted. Operators like removing a layer could considerably decrease the resources required by the model while (potentially) preserving its performance.

Hence, we now generalize the concept of network morphisms to also cover operators that reduce the capacity of a neural architecture. Let $T$ be an operator on some space of neural networks $\mathcal{N}(\mathcal{X})$, $p(x)$ a distribution on $\mathcal{X}$ and $\epsilon > 0$. We say $T$ is an $\epsilon$-approximate network morphism (ANM) with respect to a neural network $N^w$ with parameters $w$ iff

$$\Delta(T, N, w) := \min_{\tilde{w}} E_{p(x)} \left[ d\big(N^w(x), (TN)^{\tilde{w}}(x)\big) \right] \leq \epsilon, \tag{1}$$

for some measure of distance $d$. Obviously, every network morphism is an $\epsilon$-approximate network morphism (for every $\epsilon$) and the optimal $\tilde{w}$ is determined by the function-preserving property.

Unfortunately, one will not be able to evaluate the right hand side of Equation 1 in general since the true data distribution $p(x)$ is unknown. Therefore, in practice, we resort to its empirical counterpart $\tilde{\Delta}(T, N, w) := \min_{\tilde{w}} \frac{1}{|X_{train}|} \sum_{x \in X_{train}} d\big(N^w(x), (TN)^{\tilde{w}}(x)\big)$ for given training data $X_{train} \subset \mathcal{X}$. An approximation to the optimal $\tilde{w}$ can be found with the same algorithm as for training, e.g., SGD. This approach is akin to knowledge distillation (Hinton et al., 2015). We simply use categorical crossentropy as a measure of distance.

As retraining the entire network via distillation after applying an ANM is still very expensive, we further reduce computational costs as follows: in cases where the operators only affect some layers in the network, e.g., the layer to be removed as well as its immediate predecessor and successor layers, we let $TN$ first inherit all weights of $N$ except the weights of the affected layers. We then freeze the weights of unaffected layers and train only the affected weights for a few epochs.

In our experiments, we employ the following ANM's: (i) remove a randomly chosen layer or a skip connection, (ii) prune a randomly chosen convolutional layer (i.e., remove $1/2$ or $1/4$ of its filters), and (iii) substitute a randomly chosen convolution by a depthwise separable convolution. We train the affected layers for 5 epochs as described above to minimize the left hand side of Equation 1. Note that these operators could easily be extended by sophisticated methods for compressing neural networks (Han et al., 2016; Cheng et al., 2018).

## 4 MULTI-OBJECTIVE NEURAL ARCHITECTURE SEARCH

In this section, we propose a **L**amarckian **E**volutionary algorithm for **M**ulti-**O**bjective **N**eural **A**rchitecture **DE**sign, dubbed LEMONADE. We refer to Figure 1 for an illustration as well as Algorithm 1 in the appendix. LEMONADE aims at solving the multi-objective optimization problem $\min_{N \in \mathcal{N}} \mathfrak{f}(N)$ with a suitable space of neural networks $\mathcal{N}$ and the $m + n$ dimensional objective function $\mathfrak{f}(N) = (f_{exp}(N), f_{cheap}(N))^\top \in \mathbb{R}^m \times \mathbb{R}^n$, whose first component $f_{exp}(N) \in \mathbb{R}^m$ denotes expensive-to-evaluate objectives (such as the validation error or a measure of compatibility with respect to some hardware which might only be obtainable by expensive simulation) and its other components $f_{cheap}(N) \in \mathbb{R}^n$ denote cheap-to-evaluate objectives (such as model size) that one also tries to minimize.

LEMONADE maintains a population $\mathcal{P}$ of parent networks, which we choose to comprise all non-dominated networks with respect to $\mathfrak{f}$, i.e., the current approximation of the Pareto front[2]. In every generation of LEMONADE, we sample parent networks with respect to some distribution on the population and generate child networks by applying network operators (which act as mutations) to them ( see Section 3 and Appendix A.1 for details).

[1] If one would decrease the network's capacity, the function-preserving property could not be satisfied.

[2] One could also include some dominated architectures in the population to increase diversity, but we do not consider this in this work.

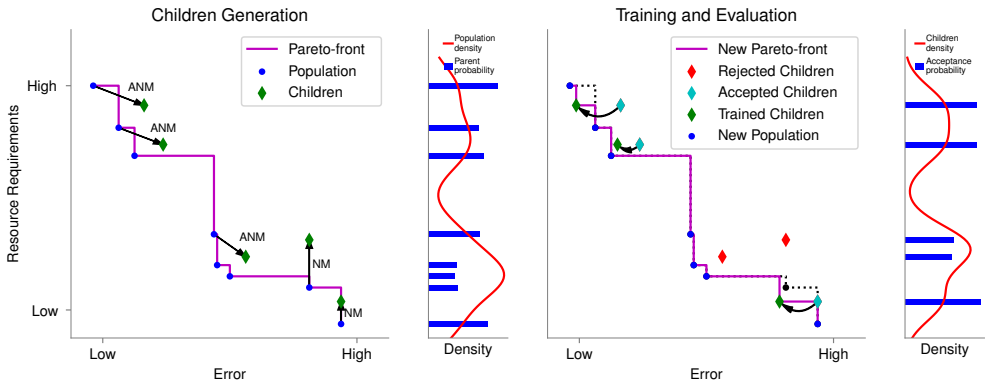


Figure 1: Conceptual illustration of LEMONADE. (Left) LEMONADE maintains a population of trained networks that constitute a Pareto-front in the multi-objective space. Parents are selected inversely proportional to their density. Children are generated by mutation operators with Lamarckian inheritance that are realized by network morphisms (NM) and approximate network morphisms (ANM). NM operators generate children with the same initial error as their parent. In contrast, children generated with ANM operators may incur a (small) increase in error compared to their parent. However, their initial error is typically still very small. (Right) Only a subset of the generated children is accepted for training: children with resource requirements different from other children are more likely to be selected for training. After training, the performance of the children is evaluated and the population and the corresponding Pareto-front is updated.

We exploit that $f_{cheap}$ is cheap to evaluate in order to bias our sampling process of children towards areas of $\mathfrak{f}$ that are sparsely populated. First, we generate a large amount of children. For selecting the parents of these children, we define a sampling distribution $p_{\mathcal{P}}(N)$ on elements $N \in \mathcal{P}$ that is inversely proportional to a density estimate of the population in the objective space $f_{cheap}$ (which can, e.g., be obtained by a kernel density estimator). Since children have similar resource requirements as their parents (network morphisms do not change architectures drastically), this sampling distribution of the parents is more likely to also generate children in less dense regions of $f_{cheap}$. We now only evaluate the cheap objectives $f_{cheap}$ of $\mathfrak{f}$ for all generated children and apply the same sampling process for selecting only a small subset of children (i.e., sampling with probabilities inversely proportional to the density estimation). Solely this subset is then trained to evaluate the expensive objectives $f_{exp}$. We thereby evaluate the cheap objectives $f_{cheap}$ of $\mathfrak{f}$ many times in order to end up with a diverse set of children, but evaluate the expensive part, $f_{exp}$, only a few times. By this two-stage sampling strategy we generate, train, and evaluate more children that have the potential to fill gaps in $\mathfrak{f}$. Finally, we compute the Pareto front from the current generation and the newly generated children, yielding the next generation. The described procedure is repeated until a maximum number of generation is reached (100 in our experiments). We refer to Appendix A.2 and A.3 for a more detailed discussion and empirical comparison of this sampling strategy to uniform sampling, respectively.

## 5 Experiments

We present results for LEMONADE on searching neural architectures for CIFAR-10 in Section 5.1. In Section 5.2, we empirically evaluate the quality of convolutional cells discovered by LEMONADE for CIFAR-10 on a different dataset, namely ImageNet64x64 (Chrabaszcz et al., 2017).

### 5.1 Experiments on CIFAR-10

We aim at solving the following multi-objective problem: beside validation error as first objective to be minimized, we use $f_{cheap}(N) = \log(\#params(N))$ as a second objective as a proxy for memory consumption[3]. The set of operators $\mathcal{T}$ we consider in the experiments are the three network morphism operators (insert convolution, insert skip connection, increase number of filters), as well as the three approximate network morphism operators (remove layer, prune filters, replace layer)

[3]One could also easily use other resources, such as inference time or multiply-add operations.

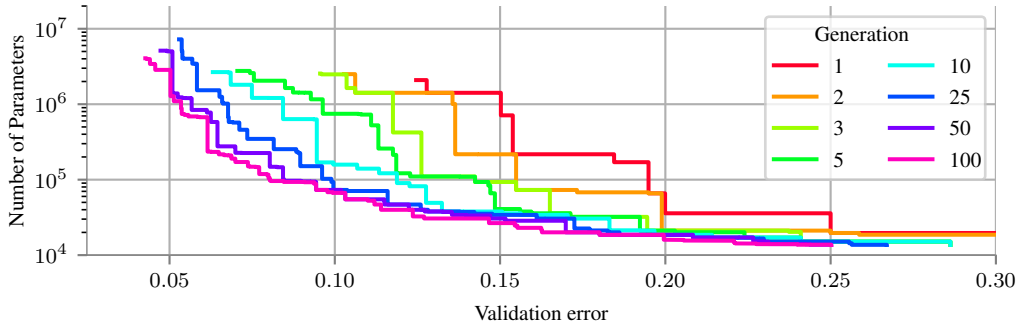


Figure 2: Progress of the Pareto-front of LEMONADE during architecture search for Search Space I. The Pareto-front gets more and more densely settled over the course of time. Very large models found (e.g., in generation 25) are discarded in a later generation as smaller, better ones are discovered. Note: generation 1 denotes the generation after one iteration of LEMONADE.

described in Appendix A.1 and Section 3, respectively. The operators are sampled uniformly at random to generate children. We evaluate LEMONADE with two different search spaces. For both spaces, LEMONADE was run for 100 generations, which took approximately seven days on eight GPUs in a straight-forward parallel asynchronous fashion. In every generation, 200 children were proposed and 40 of them accepted. In total, $4000 = 100 \cdot 40$ models were trained and evaluated during the evolutionary process. The search spaces are described below.

**Search Space I (SS-I)** This search space corresponds to searching for an entire architecture (rather than cells). We chose the number of operators that are applied to generate children uniformly from $\{1,2,3\}$. LEMONADE's Pareto front was initialized to contain four simple convolutional networks with 15 000 to 400 000 parameters and relatively large validation errors of $30 - 50\%$; see Appendix A.4 for details. LEMONADE natively handles this unconstrained, arbitrary large search space, whereas other methods are by design restricted a-priori to relativly small search spaces (Bender et al., 2018; Liu et al., 2018b) or are initialized with already well-performing networks (Cai et al., 2018b; Dong et al., 2018).

The progress of LEMONADE for this search space is visualized in Figure 2. The Pareto-front improves over time, reducing the validation error while covering a regime where the number of parameters is between 10 000 and 10 000 000.

**Search Space II (SS-II)** We use LEMONADE for discovering convolutional cells that can also be transferred to other datasets as proposed by Zoph et al. (2018). We use the same scheme for building architectures from cells as Liu et al. (2017). During architecture search, our models consist of 8 cells, the number of filters in the last block is fixed as $F \in \{128, 256\}$. We only choose a single operator to generate children, but the operator is applied to all 8 occurrences of the cell in the parent architecture. The Pareto Front was again initialized with four trivial cells; see Appendix A.4 for details.

We selected 5 out of 21 cells from the Pareto-front for further experiments, covering the whole range of different-sized cells. These cells are ordered by size, i.e., Cell 0 denotes the largest cell, while Cell 21 denotes the smallest. We refer to Appendix A.6 for an illustration of the different cells. We highlight: in contrast to Zoph et al. (2018), we do not constrain the size and structure of the cells as doing so introduces additional hyperparameters and restricts the search space. As a consequence, our cells could in principle become as complex as a whole architecture.

**Results under identical training condition** We compare the performance of the discovered architectures and cells with the following baselines:

1. Different-sized versions of MobileNet V1,V2 (Howard et al., 2017; Sandler et al., 2018), adapted for CIFAR-10[4]; these are manually designed architecture aiming for small resource-consumption while retaining high predictive performance.

[4]We replaced three blocks with stride 2 with identical blocks with stride 1 to adapt the networks to the lower spatial resolution of the input. We chose the replaced blocks so that there are the same number of stride 1 blocks between all stride 2 blocks.

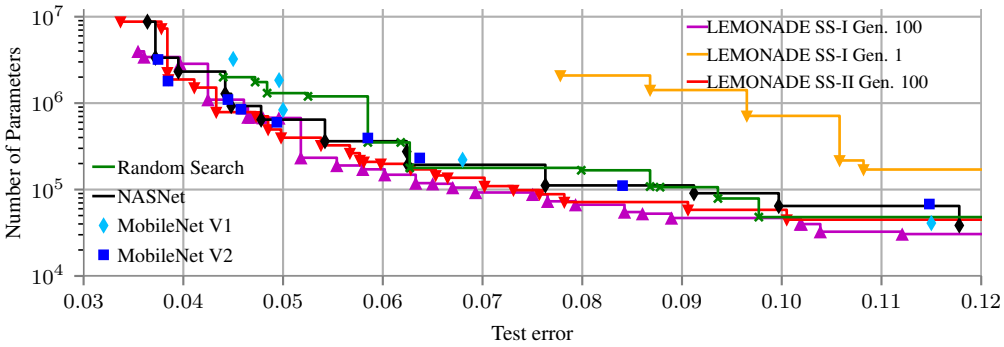


Figure 3: Performance on CIFAR-10 test data of models that have been trained under identical conditions. See also refer to Figure 6 in the appendix.

2. Different-sized NASNets (Zoph et al., 2018); NASNets are the result of neural architecture search by reinforcement learning and previously achieved state-of-the-art performance on CIFAR-10.
3. A random search baseline, where we generated random networks, trained and evaluated them and computed the resulting Pareto front (with respect to the validation data). The number and parameter range of these random networks as well as the training time (for evaluating validation performance) was exactly the same as for LEMONADE to guarantee a fair comparison.
4. Networks from generation 1 of LEMONADE. One could argue that the progress in Figure 2 is mostly due to pretrained models being trained further. To show that this is not the case, we also evaluated all models from generation 1.

We evaluated MobileNets V1, V2 with different width multiplier $\alpha$ and NASNets with different number of cells per block and different number of filters (details in Appendix A.4). We apply model augmentation (Real et al., 2018; Zoph et al., 2018) to scale up the size of models by varying the number of cells per block and the number of filters in the last block.

In order to ensure that differences in test error are actually caused by differences in the discovered architectures rather than different training conditions, we retrained all architectures from scratch using exactly the same optimization pipeline with the same hyperparameters, which is described in detail in Appendix A.4. We do not use stochastic regularization techniques, such as ShakeShake (Gastaldi, 2017) or ScheduledDropPath (Zoph et al., 2018) in this experiment as they are not applicable to all networks out of the box.

| Method | Params | Error (%) |
|---|---|---|
| DPP-Net | 0.5M | 4.62 |
| LEMONADE Cell 9 | 0.5M | 4.57 |
| DPP-Net | 1.0M | 4.78 |
| LEMONADE Cell 9 | 1.1M | 3.69 |
| NASNet V2 | 3.3M | 2.65 |
| ENAS | 4.6M | 2.89 |
| PLNT | 5.7M | 2.49 |
| LEMONADE Cell 2 | 4.7M | 3.05 |
| DPP-Net | 11.4M | 4.36 |
| PLNT | 14.3M | 2.30 |
| LEMONADE Cell2 | 13.1M | 2.58 |

Table 1: Comparison between LEMONADE and other NAS methods on CIFAR-10 for different-sized models..

The results are illustrated in Figure 3 (see also Figure 6). The best-performing model found by LEMONADE in SS-I achieves just above $3.5\%$ test error while having only 4M parameters and less than $3.4\%$ with SS-II and $8.7M$ parameters. In both spaces, LEMONADE discovers architectures with a test error of approximately $4.2\%$ while only having $1M$ parameters. The Pareto-front of LEMONADE for both spaces is competitive with MobileNet V1, V2 and NASNet for large networks and outperforms them in the regime of less than 500k parameters. We conclude that using multiple objectives during architecture search is beneficial and that using multiple architectures is superior to using just a single model which is scaled by varying a few model hyperparameters. We highlight that this result has been achieved based on using only 56 GPU days for LEMONADE compared to 2000 in Zoph et al. (2018) and with a significantly more complex Search Space I (since the entire architecture was optimized and not only a convolutional cell). LEMONADE also consistently outperforms the random search baseline and a significant improvement from generation 1 to 100 can be observed. Compare also Table 2 in the appendix for detailed numbers.

**Comparison to published results** In the last paragraph we compared different models when trained with the exact same data augmentation and training pipeline. We now also briefly compare

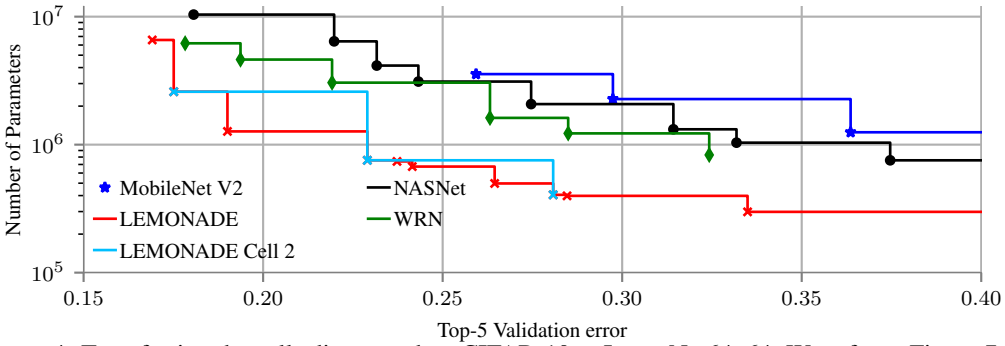


Figure 4: Transferring the cells discovered on CIFAR-10 to ImageNet64x64. We refer to Figure 7 for top-1 error. A single Cell, namely Cell 2, outperforms all baselines. Utilizing all 5 Cells (red line) further improves the results.

LEMONADE's performance to results stated in the literature.We apply two widely used methods to improve results over the last paragraph: (i) cells are employed within a well performing, hand crafted architecture, meaning one replaces repeating building blocks in the architecture with discovered cells Cai et al. (2018b); Dong et al. (2018) and (ii) using stochastic regularization techniques such as ScheduledDropPath during training Zoph et al. (2018); Pham et al. (2018); Cai et al. (2018b). For LEMONADE, we employ the discovered cells within the Shake-Shake architecture and also use Shake-Shake regularization (Gastaldi, 2017).

We compare LEMONADE with the state of the art single-objective methods by Zoph et al. (2018) (NASNet), Pham et al. (2018) (ENAS) and Cai et al. (2018b) (PLNT), as well as multi-objective (Dong et al., 2018) (DPP-Net). The results are summarized in Table 1. LEMONADE is on par or outperforms DPP-Net across all parameter regimes. As all other methods solely optimize for accuracy, they do not evaluate models with few parameters. However, also for larger models, LEMONADE is competitive to methods that require significantly more computational resources (Zoph et al., 2018) or start their search with non-trivial architectures (Cai et al., 2018b; Dong et al., 2018). ENAS achieves similar performance with a comparably complex search space, but it does not solve the multi-objective problem.

### 5.2 Transfer to ImageNet64x64

To study the transferability of the discovered cells to a different dataset (without having to run architecture search itself on the target dataset), we built architectures suited for ImageNet64x64 (Chrabaszcz et al., 2017) based on five cells discovered on CIFAR-10. We again vary (1) the number of cells per block and (2) the number of filters in the last block. We compare against different sized MobileNets V2, NASNets and Wide Residual Networks (WRNs) (Zagoruyko & Komodakis, 2016). For direct comparability, we again train all architectures in the same way.

In Figure 4, we plot the Pareto Front from all cells combined, as well as the Pareto Front from a single cell, Cell 2, against the baselines. Both clearly dominate NASNets, WRNs and MobileNets V2 over the entire parameter range, showing that a multi-objective search again is beneficial.

## 6 Conclusion

We have proposed LEMONADE, a multi-objective evolutionary algorithm for architecture search. The algorithm employs a Lamarckian inheritance mechanism based on (approximate) network morphism operators to speed up the training of novel architectures. Moreover, LEMONADE exploits the fact that evaluating objectives such, as the performance of a neural networks is orders of magnitude more expensive than evaluating, e.g., a model's number of parameters. Experiments on CIFAR-10 and ImageNet64x64 show that LEMONADE is able to find competitive models and cells both in terms of accuracy and of resource efficiency.

We believe that using more sophisticated concepts from the multi-objective evolutionary algorithms literature and using other network operators (e.g., crossovers and advanced compression methods) could further improve LEMONADE's performance in the future.

## REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *ICLR*, 2017a.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating Neural Architecture Search using Performance Prediction. In *arXiv:1705.10823 [cs]*, 2017b.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, 2018.

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *arXiv preprint*, 2017.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018a.

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-Level Network Transformation for Efficient Architecture Search. In *International Conference on Machine Learning*, June 2018b.

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

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Process. Mag.*, 35(1): 126–136, 2018.

François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017. URL `http://arxiv.org/abs/1707.08819`.

Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001. ISBN 047187339X.

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

T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *European Conference on Computer Vision*, 2018.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Simple And Efficient Architecture Search for Convolutional Neural Networks. In *NIPS Workshop on Meta-Learning*, 2017.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *AutoML: Methods, Systems, Challenges*, 2018. URL https://www.automl.org/book/.

Xavier Gastaldi. Shake-shake regularization. *ICLR 2017 Workshop*, 2017.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CVPR*, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016b.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 2012.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint*, abs/1503.02531, 2015. URL https://arxiv.org/abs/1503.02531.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. In *arXiv:1704.04861 [cs]*, April 2017.

Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. 2016.

Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Condensenet: An efficient densenet using learned group convolutions. *arXiv preprint*, 2017a.

Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *CVPR*, 2017b.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360 [cs]*, 2016.

Y.-H. Kim, B. Reddy, S. Yun, and Ch. Seo. NEMO: Neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy. In *ICML'17 AutoML Workshop*, 2017.

A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter. Learning curve prediction with Bayesian neural networks. In *International Conference on Learning Representations (ICLR) 2017 Conference Track*, April 2017.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. In *arXiv:1712.00559 [cs, stat]*, December 2017.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical Representations for Efficient Architecture Search. In *ICLR*, 2018a.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *arXiv:1806.09055*, 2018b.

I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations (ICLR) 2017 Conference Track*, April 2017.

Kaisa Miettinen. *Nonlinear Multiobjective Optimization*. Springer Science & Business Media, 1999.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving Deep Neural Networks. In *arXiv:1703.00548 [cs]*, March 2017.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, 2018.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc V. Le, and Alex Kurakin. Large-scale evolution of image classifiers. *ICML*, 2017.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized Evolution for Image Classifier Architecture Search. In *arXiv:1802.01548 [cs]*, February 2018.

Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. URL `http://arxiv.org/abs/1801.04381`.

Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. *arXiv preprint*, 2016.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *CVPR*, 2016.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint*, 2018.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. *arXiv preprint*, 2016.

Lingxi Xie and Alan Yuille. Genetic CNN. In *ICCV*, 2017.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint*, 2016.

Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint*, abs/1710.09412, 2017. URL `http://arxiv.org/abs/1710.09412`.

Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical Network Blocks Design with Q-Learning. *AAAI*, 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Conference on Computer Vision and Pattern Recognition*, 2018.

# A APPENDIX

## A.1 NETWORK MORPHISMS

Let $\mathcal{N}(\mathcal{X})$ denote a space of neural networks, where each element $N \in \mathcal{N}(\mathcal{X})$ is a mapping from $\mathcal{X} \subset \mathbb{R}^n$ to some other space, e.g., mapping images to labels. A network operator $T : \mathcal{N}(\mathcal{X}) \times \mathbb{R}^k \to \mathcal{N}(\mathcal{X}) \times \mathbb{R}^j$ maps a neural network $N^w \in \mathcal{N}(\mathcal{X})$ with parameters $w \in \mathbb{R}^k$ to another neural network $(TN)^{\tilde{w}} \in \mathcal{N}(\mathcal{X}), \tilde{w} \in \mathbb{R}^j$. We discuss one specific class of network operators, namely *network morphisms*, below.

Chen et al. (2015) introduced two function-preserving operators for deepening and widening a neural network. Wei et al. (2016) built up on this work, dubbing function-preserving operators on neural network *network morphisms*. Formally, a network morphism is a network operator satisfying the network morphism equation:

$$N^w(x) = (TN)^{\tilde{w}}(x) \text{ for every } x \in \mathcal{X}, \tag{2}$$

i.e., $N^w$ and $(TN)^{\tilde{w}}$ represent the same function. This can be achieved by properly initializing $\tilde{w}$.

### A.1.1 NETWORK MORPHISM OPERATORS USED IN OUR WORK

We list the operators we use in our work below and describe how they can be formulated as a network morphism afterwards. Details are given afterwards.

1. Inserting a Conv-BatchNorm-ReLU block. We initialize the convolution to be an identity mapping, as done by Chen et al. (2015) ("Net2DeeperNet"). Offset and scale of BatchNormalization are initialized to be the (moving) batch mean and (moving) batch variance. Since the ReLU activation is idempotent, nothing has to be done here.
2. Increase the number of filters. This operator requires the layer to be altered to have a single subsequent convolutional layer, whose parameters are padded with 0's. Alternatively, one could use the "Net2WiderNet" operator by Chen et al. (2015).
3. Add a skip connection. We allow skip connection either by concatenation (Huang et al., 2017b) or by addition (He et al., 2016a). In the former case, we proceed as in the case of increasing the number of filters. In the latter case, we do not simply add two outputs $x$ and $y$ but rather use a convex combination $(1 - \lambda)x + \lambda y$ and with a learnable parameter $\lambda$ initialized as 0 (assuming $x$ is the original output and $y$ the output of an earlier layer).

### A.1.2 DETAILS ON NETWORK MORPHISMS

**Network morphism Type I.** Let $N_i^{w_i}(x)$ be some part of a neural architecture $N^w(x)$, e.g., a layer or a subnetwork. We replace $N_i^{w_i}$ by

$$\tilde{N}_i^{\tilde{w}_i}(x) = AN_i^{w_i}(x) + b, \tag{3}$$

with $\tilde{w}_i = (w_i, A, b)$. The network morphism equation (2) then holds for $A = \mathbf{1}, b = \mathbf{0}$. This morphism can be used to add a fully-connected or convolutional layer, as these layers are simply linear mappings. Chen et al. (2015) dubbed this morphism "Net2DeeperNet". Alternatively to the above replacement, one could also choose

$$\tilde{N}_i^{\tilde{w}_i}(x) = C(AN_i^{w_i}(x) + b) + d, \tag{4}$$

with $\tilde{w}_i = (w_i, C, d)$. $A, b$ are fixed, non-learnable. In this case, network morphism Equation (2) holds if $C = A^{-1}, d = -Cb$. A Batch Normalization layer (or other normalization layers) can be written in the above form: $A, b$ represent the batch statistics and $C, d$ the learnable scaling and shifting.

**Network morphism Type II.** Assume $N_i^{w_i}$ has the form $N_i^{w_i}(x) = Ah^{w_h}(x) + b$ for an arbitrary function $h$. We replace $N_i^{w_i}$, $w_i = (w_h, A, b)$, by

$$\tilde{N}_i^{\tilde{w}_i}(x) = \begin{pmatrix} A & \tilde{A} \end{pmatrix} \begin{pmatrix} h^{w_h}(x) \\ \tilde{h}^{w_{\tilde{h}}}(x) \end{pmatrix} + b \tag{5}$$

with an arbitrary function $\tilde{h}^{w_{\tilde{h}}}(x)$. The new parameters are $\tilde{w}_i = (w_i, w_{\tilde{h}}, \tilde{A})$. Again, Equation (2) can trivially be satisfied by setting $\tilde{A} = 0$. We can think of two modifications of a neural network that can be expressed by this morphism: Firstly, a layer can be widened (i.e., increasing the number of units in a fully connected layer or the number of channels in a CNN - the Net2WiderNet transformation of Chen et al. (2015)). Let $h(x)$ be the layer to be widened. For example, we can then set $\tilde{h} = h$ to simply double the width. Secondly, skip-connections by concatenation as used by Huang et al. (2016) can also be expressed. If $h(x)$ itself is a sequence of layers, $h(x) = h_n(x) \circ \cdots \circ h_0(x)$, then one could choose $\tilde{h}(x) = x$ to realize a skip from $h_0$ to the layer subsequent to $h_n$.

**Network morphism Type III.** By definition, every idempotent function $N_i^{w_i}$ can simply be replaced by

$$N_i^{(w_i, \tilde{w}_i)} = N_i^{\tilde{w}_i} \circ N_i^{w_i} \tag{6}$$

with the initialization $\tilde{w}_i = w_i$. This trivially also holds for idempotent functions without weights, e.g., ReLU.

**Network morphism Type IV.** Every layer $N_i^{w_i}$ is replaceable by

$$\tilde{N}_i^{\tilde{w}_i}(x) = \lambda N_i^{w_i}(x) + (1 - \lambda) h^{w_h}(x), \quad \tilde{w}_i = (w_i, \lambda, w_h) \tag{7}$$

with an arbitrary function $h$ and Equation (2) holds if the learnable parameter $\lambda$ is initialized as 1. This morphism can be used to incorporate any function, especially any non-linearity. For example, Wei et al. (2016) use a special case of this operator to deal with non-linear, non-idempotent activation functions. Another example would be the insertion of an additive skip connection, which were proposed by He et al. (2016a) to simplify training: If $N_i^{w_i}$ itself is a sequence of layers, $N_i^{w_i} = N_{i_n}^{w_{i_n}} \circ \cdots \circ N_{i_0}^{w_{i_0}}$, then one could choose $h(x) = x$ to realize a skip from $N_{i_0}^{w_{i_0}}$ to the layer subsequent to $N_{i_n}^{w_{i_n}}$.

Note that every combination of network morphisms again yields a network morphism. Hence, one could, for example, add a block "Conv-BatchNorm-ReLU" subsequent to a ReLU layer by using Equations (3), (4) and (6).

### A.2 Sampling Strategy in LEMONADE

We provide details on the procedure employed in LEMONADE for sampling parents and selecting children for training. LEMONADE first computes a kernel density estimator $p_{KDE}$ on $\{f_{cheap}(N) | N \in \mathcal{P}\}$. Note that we explicitly compute the KDE with respect to the cheap-to-evaluate function $f_{cheap}$ rather than $\mathfrak{f}$ as this allows to evaluate $p_{KDE}(f_{cheap}(N))$ very quickly. To exploit this, we first generate a larger number $n_{pc}$ of proposal children $\mathbf{N}_{pc}^c = \{N_1^c, \ldots, N_{n_{pc}}^c\}$ by applying network operators, where the parent $N$ for each child is sampled according to the distribution

$$p_{\mathcal{P}}(N) = \frac{c}{p_{KDE}(f_{cheap}(N))}, \tag{8}$$

with a normalization constant $c = 1/\big(\sum_{N \in \mathcal{P}} p_{KDE}(f_{cheap}(N))\big)$. Afterwards, we accept $n_{ac}$ children $\mathbf{N}_{ac}^c \subset \mathbf{N}_{pc}^c$. These accepted children are again sampled in an anti-proportional manner according to

$$p_{\mathcal{P}}(N^c) = \frac{\hat{c}}{p_{KDE}(f_{cheap}(N^c))}, \tag{9}$$

with $\hat{c}$ being another normalization constant. Only these accepted children are trained and evaluated according to $f_{exp}$, and used to update the Pareto front $\mathcal{P}$.

### A.3 Baseline experiment with uniform sampling instead of KDE

In order to evaluate the importance of the sampling strategy employed in LEMONADE, we compare LEMONADE to a version of LEMONADE employing uniform random sampling of parents and uniform selection of children to be trained and evaluated. The resulting Pareto-fronts after $n_{gen} = 32$ generations are illustrated in Figure 5. The most striking difference is that LEMONADE explores architectures with more than one million parameters more thoroughly than LEMONADE (uniform), which samples mostly architectures with less than one million parameter. The latter can be explained by the property that most points on the Pareto-front lie in this parameter range and, since those act as

**Algorithm 1** LEMONADE

**function** LEMONADE ($\mathcal{P}_0, \mathfrak{f}, \mathcal{T}, n_{gen}, n_{pc}, n_{ac}$ )
  $\mathcal{P} \leftarrow \mathcal{P}_0$
  **for** $i \leftarrow 1, \ldots, n_{gen}$ **do**
    $p_{KDE} \leftarrow KDE\big(\{f_{cheap}(N) | N \in \mathcal{P}\}\big)$
    Compute parent distribution $p_{\mathcal{P}}$ (Eq. 8)
    $\mathbf{N}^c_{pc} \leftarrow GenerateChildren(\mathcal{P}, p_{\mathcal{P}}, n_{pc}, \mathcal{T})$
    Compute child distribution $p_{child}$ (Eq. 9)
    $\mathbf{N}^c_{ac} \leftarrow AcceptSubSet(\mathbf{N}^c_{pc}, p_{child}, n_{ac})$
    Train all $N^c \in \mathbf{N}^c_{ac}$
    Update $\mathcal{P}$ with $\mathbf{N}^c_{ac}$ according to $\mathfrak{f}$
  **end for**
  **return** $\mathcal{P}$
**end function**

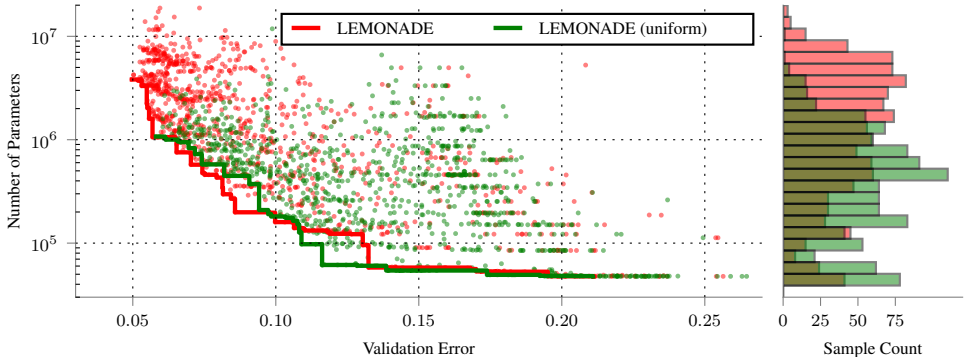


Figure 5: Pareto-front on validation data after 32 generations of architecture search as well as all sampled architectures (color-coded) for LEMONADE and its baseline version with uniform sampling. Moreover, a histogram of the number of parameters of all sampled models is shown on the right.

parents, most children will also lie close to this parameter range. In contrast, LEMONADE samples points from the Pareto-front inversely to their density and thus, elements at the the extremes of the Pareto-front will more often act as parents and will generate more offspring in the parameter ranges not yet covered by the Pareto-front.

### A.4 Experimental Details

All experiments were run on Nvidia Titan X GPUs, with code implemented in Keras (Chollet et al., 2015) and TensorFlow (Abadi et al., 2015) backend.

#### A.4.1 Detailed description of Search Space I

All four initial networks had the following structure: three Conv-BatchNorm-ReLU blocks with intermittent Max-Pooling, followed by a global average pooling and a fully-connected layer with softmax activation. The networks differ in the number of channels in the convolutions, and for further diversity two of them used depthwise-separable convolutions. The models had 15 000, 50 000, 100 000 and 400 000 parameters, respectively.

We restricted the space of neural architectures such that every architecture must contain at least 3 (depthwise separable) convolutions with a minimum number of filters, which lead to a lower bound on the number of parameters of approximately 10 000.

The network operators implicitly define the search space, we do not limit the size of discovered architectures.

#### A.4.2 DETAILED DESCRIPTION OF SEARCH SPACE II

As done by Liu et al. (2017), we use $k$ blocks composed of $n$ cells each with stride 1 and $k-1$ cells in between the blocks with stride 2 to reduce the spatial resolution, resulting in architectures being composed of $k \cdot n + (k-1)$ cells. The cells are simply stacked sequentially; the input for each cells is the concatenated output of the previous two cells. We set $k = 3, n = 2$ during architecture search, that is we use 8 cells in total. The number of filters in the first block is fixed as either 32 or 64 and doubled whenever the spatial resolution is halved, resulting in $F \in \{128, 256\}$ filters in the last block.

The Pareto Front was initialized with four cells: the first two cells consist of a single convolutional layer (followed by BatchNorm and ReLU) with $F = 128$ and $F = 256$ filters in the last block, respectively. The other two cells consist of a single depthwise separable convolution (followed by BatchNorm and ReLU), again with either $F = 128$ or $F = 256$ filters.

#### A.4.3 DETAILS ON VARYING MOBILENETS, NASNETS AND MODEL AUGMENTATION

We evaluated MobileNets V1 and V2 with different width multiplier $\alpha \in \{0.1, 0.2, \ldots, 1.2\}$ and NASNets with different number of cell per block ($\in \{2, 4, 6, 8\}$) and different number of filters ($\in \{96, 192, 384, 768, 1536\}$). We apply "model augmentation" to models built from cells as done by Real et al. (2018); Zoph et al. (2018), i.e., we scale the size of models built from cells by varying the number of cells $n \in \{2, 4, 6, 8\}$ per block and the number of filters $F \in \{64, 128, 256, 512\}$ in the last block. (Recall we used $n = 2, F \in \{128, 256\}$ during architecture search.)

#### A.4.4 EXPERIMENTS IN SECTION 5.1:

We apply the standard data augmentation scheme described by Loshchilov & Hutter (2017), as well as the recently proposed methods mixup (Zhang et al., 2017) and Cutout (Devries & Taylor, 2017). The training set is split up in a training (45.000) and a validation (5.000) set for the purpose of architecture search. We use weight decay ($5 \cdot 10^{-4}$) for all models. We use batch size 64 throughout all experiments. During architecture search as well as for generating the random search baseline, all models are trained for 20 epochs using SGD with cosine annealing (Loshchilov & Hutter, 2017), decaying the learning rate from $0.01$ to $0$. For evaluating the test performance, all models are trained from scratch on the training and validation set with the same setup as described above except for 1) we train for 600 epochs and 2) the initial learning rate is set to $0.025$. While searching for convolutional cells on CIFAR-10, `LEMONADE` also ran for approximately 56 GPU days. However, there were no significant changes in the Pareto-front after approximately 24 GPU days. The training setup (both during architecture search and final evaluation) is exactly the same as before. We present a detailed comparison of `LEMONADE`, MobileNet V1, MobileNet V2 and NASNet in Table 2.

#### A.4.5 EXPERIMENTS IN SECTION 5.2:

The training setup on ImageNet64x64 is identical to Chrabaszcz et al. (2017).

### A.5 ADDITIONAL RESULTS

Below we plot results from additional experiments.

### A.6 DISCOVERED CELLS

Below we illustrate a few cells found by `LEMONADE` on CIFAR-10 which were then transferred to ImageNet64x64.

| MODEL | PARAMS | ERROR (%) |
|---|---|---|
| MOBILENET | 40K | 11.5 |
| MOBILENET V2 | 68K | 11.5 |
| NASNET | 38K | 12.0 |
| RANDOM SEARCH | 48K | 10.0 |
| LEMONADE | 47K | **8.9** |
| MOBILENET | 221K | 6.8 |
| MOBILENET V2 | 232K | 6.3 |
| NASNET | 193K | 6.8 |
| RANDOM SEARCH | 180K | 6.3 |
| LEMONADE | 190K | **5.5** |
| MOBILENET | 834K | 5.0 |
| MOBILENET V2 | 850K | **4.6** |
| NASNET | 926K | 4.7 |
| RANDOM SEARCH | 1.2M | 5.3 |
| LEMONADE | 882K | **4.6** |
| MOBILENET | 3.2M | 4.5 |
| MOBILENET V2 | 3.2M | 3.8 |
| NASNET | 3.3M | 3.7 |
| RANDOM SEARCH | 2.0M | 4.4 |
| LEMONADE | 3.4M | **3.6** |

Table 2: Comparison between LEMONADE (SS-I), Random Search, NASNet, MobileNet and MobileNet V2 on CIFAR-10 for different model sizes.

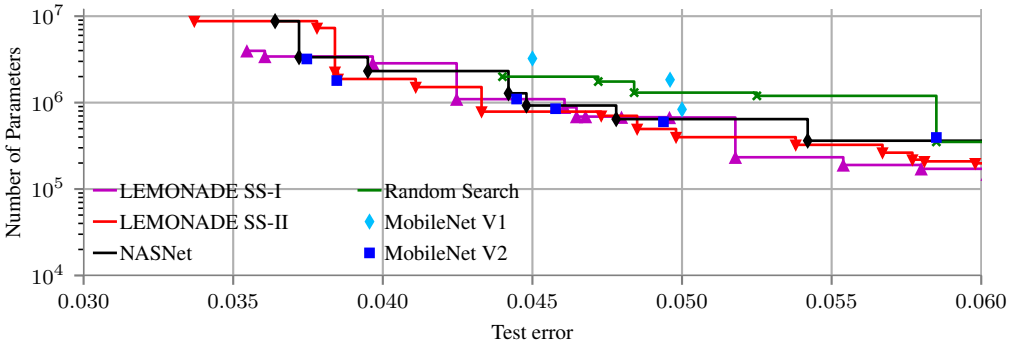


Figure 6: Comparison of LEMONADE with other NAS methods and hand crafted architectures on CIFAR-10. This plot shows the same results as Figure 3 but zoomed into the rage of errors less than 0.06%.

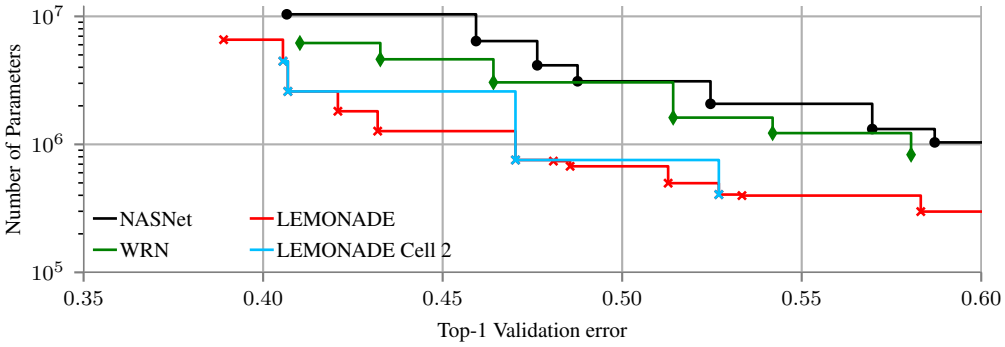


Figure 7: Transferring the cells discovered on CIFAR-10 to ImageNet64x64. Top-1 validation error.

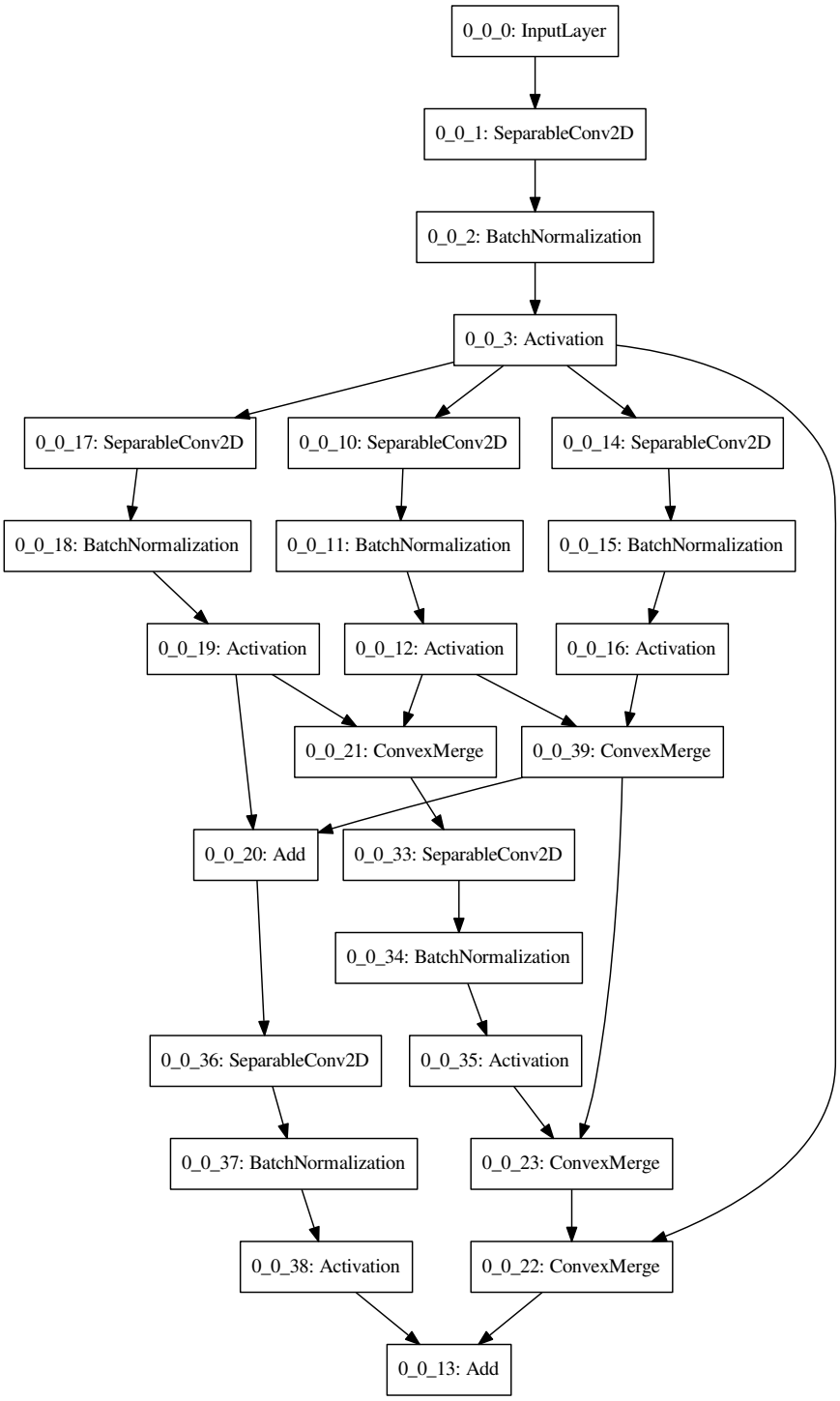


Figure 8: Cell 0. Largest discovered cell.

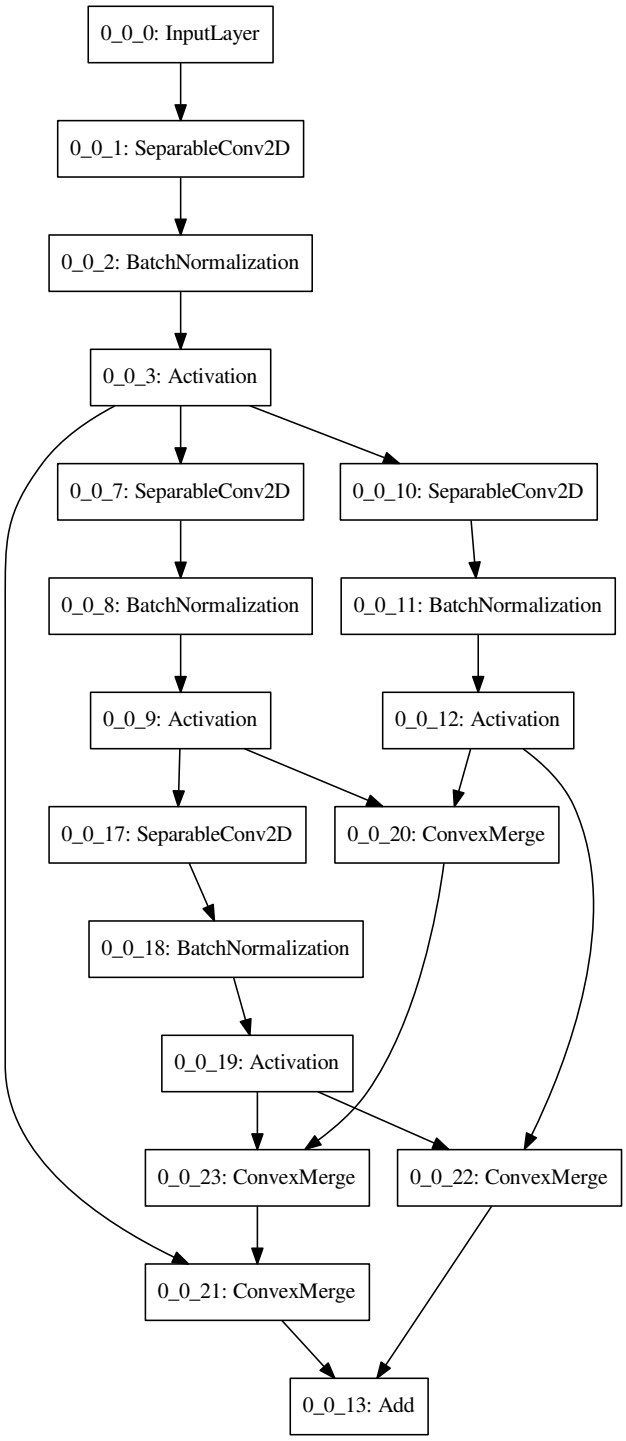


Figure 9: Cell 2

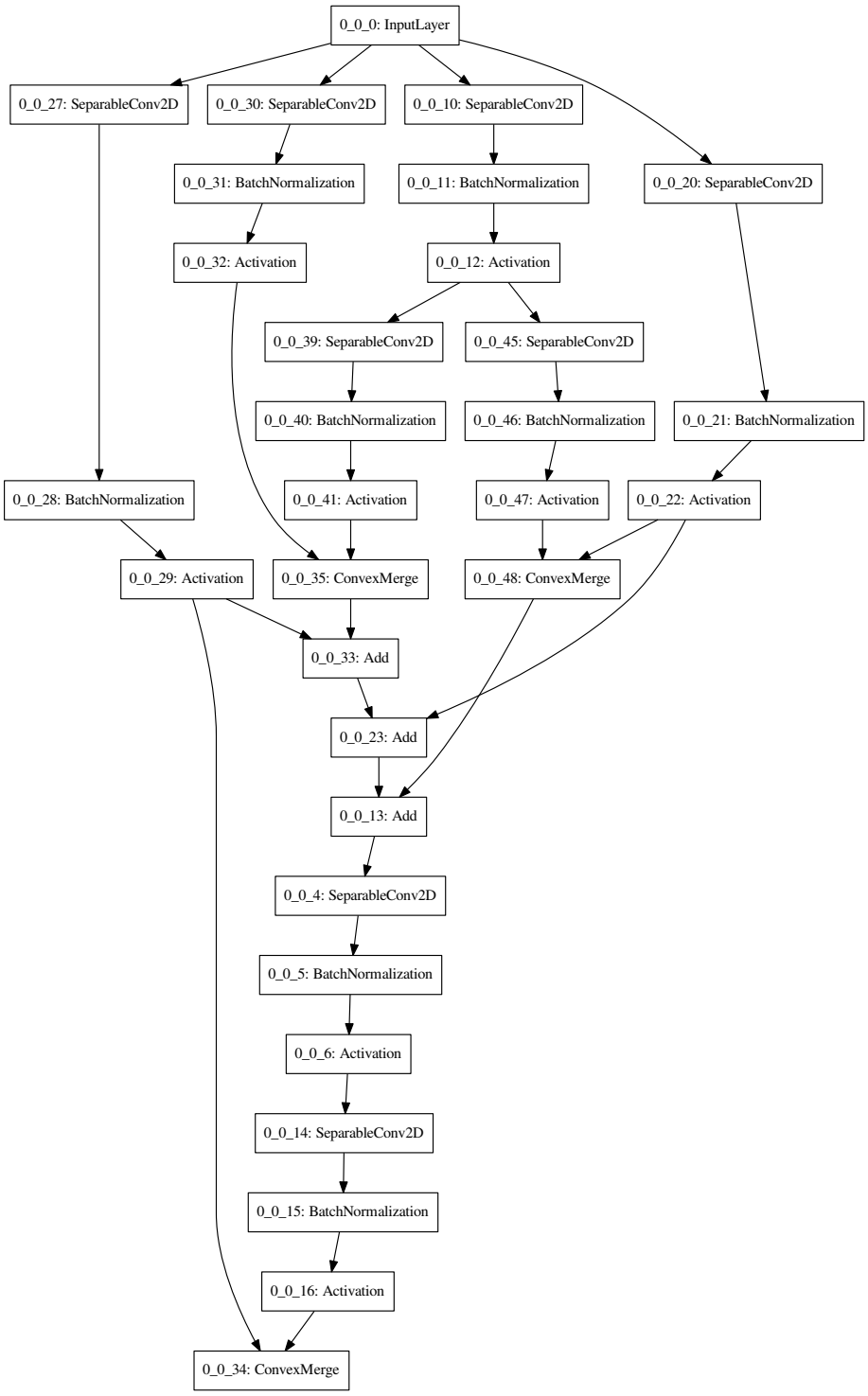


Figure 10: Cell 6

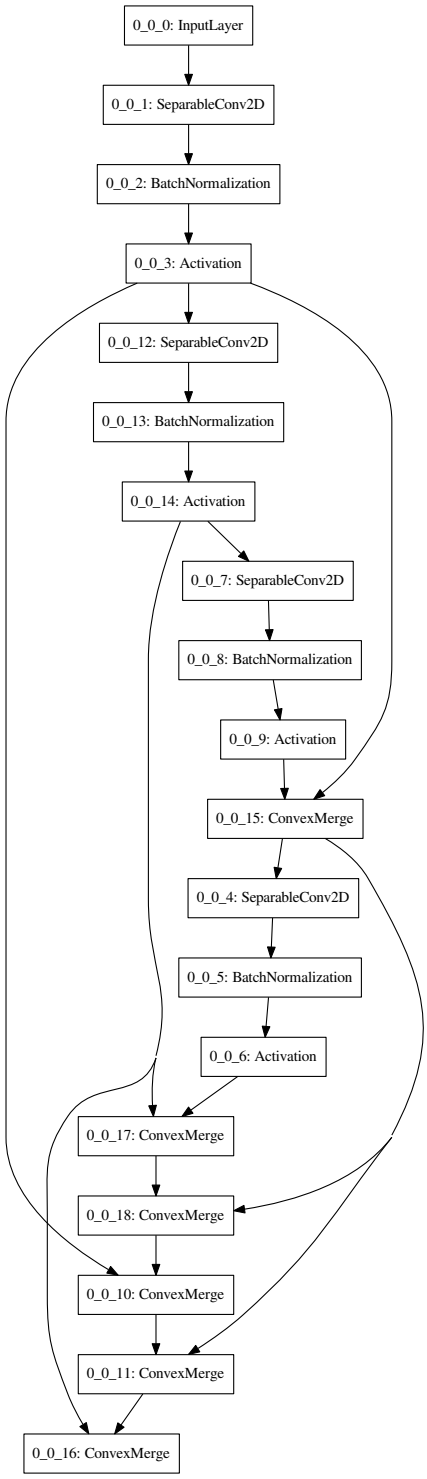


Figure 11: Cell 9

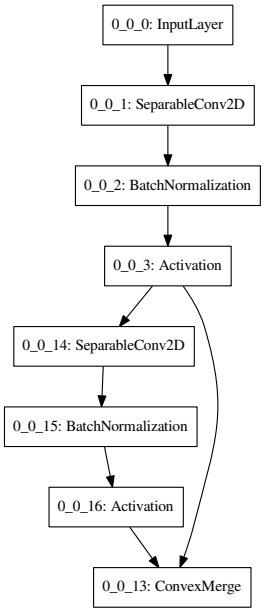


Figure 12: Cell 18

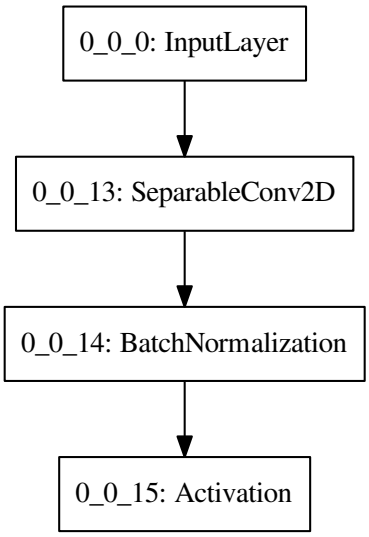


Figure 13: Cell 21. The smallest possible cell in our search space is also discovered.