Neural Architecture Search with Network Morphisms and Successive Halving

Miray Yüce [*1] **Olesya Tsapenko** [*1]

Abstract

Deep learning is proving its effectiveness on a variety of tasks every day. However this success highly depends on the hyper parameters involved in training and neural network architecture used. Both architecture design and hyper parameter optimization are costly and laborious tasks. In this work, we want to extend a previous work (Elsken et al., 2017); which uses network morphisms and hill climbing for convolutional neural network search; by adding successive halving to the search process. Moreover, we widen our baseline's search space by adding separable convolutional layers. Large scale of experiments are conducted with CIFAR-10. Additionally, we test different learning rate schedulers, and share the effects of different data augmentation combinations on a small base network. All measures show that adding successive halving is a competitive child network selection strategy with hill climbing alone, while it generates smaller models than our baseline does.

1. Introduction

In the last decade deep neural networks have garnered significant interest from researchers as they have achieved human-level or above success on difficult problems such as image classification (Krizhevsky et al., 2012; Sabour et al., 2017; Cüreşan et al.), object detection (Ren et al., 2016; Lui et al., 2016), instance segmentation (He et al.), speech recognition (Graves et al., 2013), machine translation (Ahmed et al., 2017), language modeling (Yang et al., 2018) and reinforcement learning (Silver et al., 2017). Success of these implementations depends highly on the architecture of the neural networks; in other words, the function represented by the network. To achieve state-of-the-art results, researchers have tried to improve the performance of the models by designing the architecture manually (He et al., 2015; Simonyan & Zisserman, 2015; Krizhevsky et al., 2012; Szegedy et al., 2014). However, neural architecture engineering is a time consuming and error prone task. Moreover, it is limited by the imagination and expertise of academics. Hence, a new field of research called neural architecture search emerged to find the optimal architectures.

Neural architecture search is mainly based on evolutionary methods (Real et al., 2018; 2017; Elsken et al., 2017; Jin et al., 2018) and reinforcement learning algorithms (Zoph & Le, 2017; Pham et al., 2018; Zoph et al., 2018; Baker et al., 2017). Pioneer work was very promising even though the algorithms required thousands of GPU days (Zoph et al., 2018; Real et al., 2018). Current research (Liu et al., 2018c; Pham et al., 2018; Liu et al., 2018a) is focusing on efficient neural architecture search methods, and on architectures which can perform well even if they are trained with different datasets from the same domain.

In this work we propose an expansion to Elsken et al. (2017); an evolutionary based neural architecture search method for convolutional neural networks (LeCun et al., 1998), that uses network morphisms (Wei et al., 2016) (Section 3.1) and hill climbing. The search process with hill climbing is extended with successive halving. To compare our work, we reimplement Elsken et al. (2017) as our baseline. Additionally, since depthwise separable convolutions (Chollet, 2016) speed up the inference process by decreasing the number of parameters, we propose expanding the search space by adding these layers (3.2.6). All experiments are conducted on CIFAR-10 (Krizhevsky et al.).

The results show that successive halving based methods generate models with fewer parameters and less variance, which are also competitive with our baseline in terms of final performance. Our methods SH, SHR, and SHRSep reach average test accuracy of 94.57%, 94.63% and 94.39% respectively, while NASH achieves 94.64%.

This report's organization is as follows: we first cover related work on neural architecture search in Section 2. Section 3 explains our method and different search strategies. Experiments and their results are presented in Section 4 and Section 5, respectively. Finally, in Section 6 we discuss possible future improvements.

[*]Equal contribution [1]Department of Computer Science, University of Freiburg, Freiburg, Germany. Correspondence to: Miray Yüce <yucemiray@gmail.com>, Olesya Tsapenko <olesyatsapa@mail.ru>.

2. Background and Related Work

Recent research focuses on two main approaches to handle automated neural architecture search problem. One is using reinforcement learning based methods (Zoph & Le, 2017; Cai et al., 2017; Pham et al., 2018; Zoph et al., 2018; Baker et al., 2017). Zoph & Le (2017) to train an RNN as a controller, with a policy gradient (Sutton et al., 2000) to generate model descriptions for CNN's, and cells for RNN. The resulting network of the controller is trained until convergence, and its validation accuracy is used for updating the controller's parameters.

Zoph et al. (2018) followed the same method, and introduced cells as predictions of the controllers. Cells, which are inspired from repeating motifs of state-of-the-art handcrafted CNN's, serve as building blocks of a larger network by connecting to each other. The cells, namely normal cells and reduction cells (reduces a feature maps dimensions by a factor of two), share the same architecture within their types, but have different weights. Their main contribution is that the cells found can be used for creating different networks when shifting to a larger dataset (transfer learning). However, the resulting neural architectures are still limited to these unchanging cells, and the search cannot go beyond these blocks. The cell, as a neural network, should be able to fit into and stay in the device memory during training. Also, the connections between the cells are predefined in terms of input and output layers and connection patterns; there is no search done to find better connection styles between them. Pham et al. (2018) followed a similar method, improving efficiency by sharing parameters across child models. Liu et al. (2018a) learned a surrogate function for predicting cells' architecture performance, and used sequential model based optimization to create more complex models from the search space in a progressive manner. Another similar work, Zhong et al. (2018) used Q learning, and searched for building blocks.

The second main approach to neural architecture search problem, and the closest related work to ours is that on evolutionary algorithms. Early work tried to optimize the structure and the model parameters together with evolutionary algorithms (Stanley & Miikkulainen, 2002); however, this task requires substantial resources to optimize large scale networks. Recent evolutionary based methods (Real et al., 2017; 2018; Elsken et al., 2017; 2018a) optimize parameters using gradient based methods, while using genetic algorithms to find the best architecture. Real et al. (2017) applied a series of mutations to generate child networks, and used tournament selection. Some of the mutations they proposed overlap with ours, while most of them differ, such as changing the learning rate, stride, and resetting the weights. As the population size handled was 1000, substantial amount of computational power needed to complete the search. Real et al. (2018) proposed a state-of-the-art method for finding cells using an evolutionary algorithm, in which the old models die as they get old and disappear from the population. Liu et al. (2018b) proposed a new representation scheme, called hierarchical representation, and combined it with evolutionary methods to create cell structures.

Elsken et al. (2017) combined network morphisms Wei et al. (2016); Chen et al. (2016) and hill climbing to speed up the performance prediction of novel architectures. Elsken et al. (2018a) proposed another method for finding neural architectures, which not only have low test error, but also have low resource consumption and a small number of parameters by approximating the Pareto-front of children architectures. They also searched for cells; and decreased the computational cost significantly. A similar work, Jin et al. (2018), used Bayesian optimization to guide selection of network morphism operators, in addition to a tree-structured search space and a new network-level morphism.

Another method used in neural architecture search is learning a helper network (one-shot model) for performance prediction of candidate networks. Brock et al. (2017) sampled a random architecture at each training step, and mapped weights generated by the Hyper-Net to the candidate network. The sampled architecture's performance is evaluated on validation data, and it is used for back propagation in which the helper function's parameters are optimized. Liu et al. (2018c) followed a similar method to search CNN and RNN cells, and optimized the weights of a super-graph. Moreover, the search space is subjected to continuous relaxation. This relaxation permitted to have a continuous search space, which is optimized by gradient descent. One-shot models are also useful for transfer learning.

Cai et al. (2017) experimented on simple CNN architectures using RNN's and network transformations to preserve functionality. Cai et al. (2018) also used network transformation by proposing a new method, called path level network transformation, which alter the path topology while reusing the existing weights.

3. Method

As described in Elsken et al. (2018b) neural architecture search can be explained with 3 main components. The first one is Search Space, which characterises the possible network architecture that can be found by the method. We can categorize our search space as a multi-branch network space, without any specific cell repetition (Zoph et al., 2018; Pham et al., 2018; Liu et al., 2018c; Zhong et al., 2018).

The second one is Search Strategy, which describes the method of finding new promising architectures within the search space. Our method falls into the neuro-evolutionary methods class, in which at evolutionary step promising chil-

dren architectures are searched in a population of networks, which are generated by mutating the parent network. After each evolutionary step we receive a child network to be the parent of the next generation. We use NASH from Elsken et al. (2017) for this search strategy. We will explain the search strategies that we used for our experiments later in Section 3.3.

The last component of neural architecture search is the performance estimation strategy to direct the search to high performing architectures in the search space. As training each candidate network of the population from scratch; and evaluating their performance on validation data is very expensive, performance estimation strategy is an important part of the neural architecture search. It is preferred to find architectures with high validation performance with minimum training. We compare two performance estimation strategies in our experiments, namely hill climbing and successive halving, by combining them with Network Morphisms (Section 3.1), to choose the best performing child network using SGDR and cosine annealing (Loshchilov & Hutter, 2017).

In the rest of this section, we will first explain Network Morphisms briefly in Section 3.1, then how to use them for creating a number of network operators in Section 3.2. Lastly, we present child network selection strategies in Section 3.3.

3.1. Network Morphisms

To avoid expensive training from scratch of children networks, network morphisms principle (Wei et al., 2016) can be used to transform a network to another one. Particularly, network morphisms can be used for mutating a parent network, which yields to a new child network after mutation. This novel network inherits the functionality of the parent network without any training. After this step, as the offspring takes over the complete knowledge of the parent network, it is not needed to be trained from scratch. Moreover, we can already receive a more complex and a more powerful network by training the child network for a short period of time. This eases the performance prediction of the child network substantially, which we need to for deciding whether an architecture is promising for future mutations or not. Another useful feature of network morphisms is that every combination of them results to a network morphism again.

Network morphisms can be divided into two in terms of functionality preservation, namely exact and approximate network morphisms (Elsken et al., 2017; 2018a). Exact network morphisms increase the network size and capacity, and they yield to a child network which takes over the complete knowledge from its parent. On the other hand, approximate morphisms decrease the network capacity, and they do not necessarily preserve the complete functionality.

To give more theoretical background about how network morphisms work, here we briefly describe network morphism equation in Equation 1 as it is explained in Elsken et al. (2018a), where $\mathcal{N}(\mathcal{X})$ is a set of networks defined on dataset $\mathcal{X} \subset \mathbb{R}^n$, and T is a network morphism operator which maps a network $N \in \mathcal{N}$ with parameters w to another network TN with parameters $\tilde{w}$.

$$\begin{aligned} T : \mathcal{N}(\mathcal{X}) \times \mathbb{R}^k &\rightarrow \mathcal{N}(\mathcal{X}) \times \mathbb{R}^j \\ N^w(x) = (TN)^{\tilde{w}}(x)\,, &\forall x \in \mathcal{X} \\ w \in \mathbb{R}^k \; &and \; \tilde{w} \in \mathbb{R}^j \end{aligned} \tag{1}$$

Elsken et al. (2017) proposed four exact morphisms. From practical point of view, we can summarize network morphism types as follows (Elsken et al., 2017).

- **Type I.** It can be used for adding convolutional, fully connected and batch normalization layers.
- **Type II.** It is useful for widening network layers in terms of number of units and channels, and for adding skip connections by concatenation.
- **Type III.** For idempotent functions, such as $ReLu$.
- **Type IV.** For non linearity and inserting additive skip connections.

In the next section (Section 3.2), we will show use cases of these network morphism types to create network operators.

3.2. Network Operators

We implement five exact network operators as they are proposed in Elsken et al. (2017), and construct one more; which allows using depth wise separable convolutions. For all kind of network operators we describe below, one must remember that topology of the network, order of the layers, and input and output sizes must be checked neatly.

3.2.1. Insert Convolutional Layer

Our first network operator makes an existing network deeper by adding a block of $Conv - BatchNorm - ReLu$ layers. This operator is based on morphism types 1 and 3 (Elsken et al., 2017). As any combination of network morphisms yields to another one, we can use type 1 for inserting a single convolutional layer, and then a batch normalization layer, finally we combine them with type 3 for adding non-linearity, namely a $ReLu$ activation function.

This operator is placed on a random position in the network with a restriction: a new $Conv - BatchNorm - ReLu$ block can be inserted only between other blocks, after a

$MaxPool2d$ layer, or after any type of merged layers. The reasons behind these restrictions are preserving the existing blocks, and not inserting a block after the last Dense layer. On top of these restrictions, the new convolutional layer's kernel size is sampled randomly from $[3, 5]$ to make identity mapping of weights work, and not to have large kernels which yield more parameters .

To assure preserving functionality of the newly mutated network, we initialize new layer's kernels by $id\_mapping$. In the following example, we have $kernel\_size = 3$, $\#channels = 3$. When we use these kernels for an input with 3 channels and same padding. The output is exactly the same as input of the new layer.

$$\left[\begin{bmatrix}\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}&\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}&\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}\end{bmatrix}\right.$$
$$\left[\begin{bmatrix}\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}&\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}&\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}\end{bmatrix}\right.$$
$$\left.\left[\begin{bmatrix}\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}&\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}&\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}\end{bmatrix}\right]\right]$$

After this step, we initialize batch normalization layer's (Iofe & Szegedy, 2015) parameters to keep $y = x$.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta \tag{2}$$

According to Equation 2 we should set γ equal to denominator and β to $\mathrm{E}[x]$. This will make batch normalization layer's output as the same as its input.

3.2.2. Alter Channels

The second network operator widens the network by altering the number of output channels of a convolutional layer by a factor. There is a restriction on maximum number on channels to prevent the layer from getting too large.

This operator is constructed based on network morphism type 2, and it is also a variant of Net2WiderNet (Chen et al., 2016). Increasing the number of channels corresponds to increasing the number of kernels of a convolutional layer; hence, if a layer has m output channels, after increasing the number of channels by a factor f, there will be $f \cdot m$ kernels. We only preserve m old kernels, and leave the rest as they are initialized.

Changing the number of output channels of a layer effects the next convolutional layer's input channels. To take care of this change, we adjust input size of the subsequent convolutional layer accordingly: if the subsequent layer has kernels with size (k, k, m), after altering predecessor layer's channels, its kernel size becomes $(k, k, (m \cdot f))$. We preserve the functionality by keeping original m $k \times k$ matrices as they are, and set the rest as $zero$ matrices to prevent previous layer's extra channels affecting the output.

3.2.3. Merge by Convex Combination

This network operator carries out a convex combination of two compatible layers, based on network morphism type 4, (Elsken et al., 2017), as given in Equation 3. The layers to be merged can be only $ReLu$ (the last layer of a block) or $MaxPool2d$ for the same reasons as we mentioned in Insert Convolutional Layer in Section 3.2.1. Also, the layers must share the same dimensions (C, H, W).

$$l^* = \lambda l_i + (1 - \lambda) l_j,\ where\, i \neq j \tag{3}$$

The compatible layers are combined with respect to a learnable parameter λ, which is initialized as 1.0 to keep the output of the new layer as the same as the first layer. Later, this situation can change as λ will be learned. We can define this through an abstract equation, in Equation 3 where l_i, l_j are appropriate layers, and l^* is the new layer, which is input to l_{i+1}.

This operation can also be carried out with layers which do not share the same input dimensions. Merging operation can be done after downsampling one of them as we define in Section 3.2.4.

3.2.4. Merge by Concatenation

Merge by Concatenation network operator is based on network morphism type 2. The layers to be merged can only be $ReLu$ or $MaxPool2d$ as it is for Merge by Convex Combination in Section 3.2.3.

The candidate layers are found according to equation 3.2.4 for layers l_i and l_j.

$$f \in \{1, 2, 4\}$$
$$(H_i, W_i) = f \cdot (H_j, W_j),\ where\, i \neq j \tag{4}$$

When the $f \neq 1$, the layer with larger dimensions is simply downsampled by using a $MaxPool2d$ layer to the other layer's dimensions. After dimensions match, the resulting layers are stacked together such that the output dimension of the new merging layer is l^* is $(C_i + C_j, H_i, W_j)$, and it is input to l_{i+1}:

$$l^* = concat(l_i, l_j),\ where\, i \neq j \tag{5}$$

To preserve functionality, we modify layer l_{i+1}'s weights as we do for subsequent convolutional layer in Alter Channels in Section 3.2.2.

3.2.5. Split Up Convolutional Layer

Our fifth network operator is Split Up Convolutional Layer, which is based on network morphism type 4. We split up a $Conv - BatchNorm - ReLu$ block i into two parallel $Conv - BatchNorm - ReLu$ blocks, by replacing it with two new blocks as follows:

$$block_{i_1} = \alpha \cdot block_i$$
$$block_{i_2} = (1 - \alpha) \cdot block_i$$
$$l^* = output(block_{i_1}) + output(block_{i_2})$$

The split up factor α we use is a constant 0.3. This operator splits the $Conv - BatchNorm - ReLu$ to two $Conv - BatchNorm - ReLu$ blocks, and maps the corresponding parameters to the newly created layers using the split up factor α. These two parallel blocks' outputs are then summed into newly created layer l^*.

3.2.6. Insert Separable Convolutional Layer

We can imagine inserting a depthwise separable convolution layer (Chollet, 2016) as inserting two convolutional layers in a row; first a depthwise convolutional layer, then a pointwise convolutional layer. In depthwise convolutional layer, the kernels are applied to the input channels individually, and the resulting feature maps are stacked together. After this step, a pointwise convolutional layer; which has always $kernel\_size = 1$, is applied to the stacked feature maps. Number of kernels with $kernel\_size = 1$ is equal to number of output channels.

For preserving functionality, we should take care of two $id\_mapping$ operations. Let us imagine a scenario where $kernel\_size = 3$, $\#input\_channels = 3$, $\#output\_channels = 3$, and same padding is used on input to depthwise layer. First, we initialize depthwise convolutional layer kernels as follows. This initialization gives exactly the same output as input.

$$\begin{bmatrix} \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

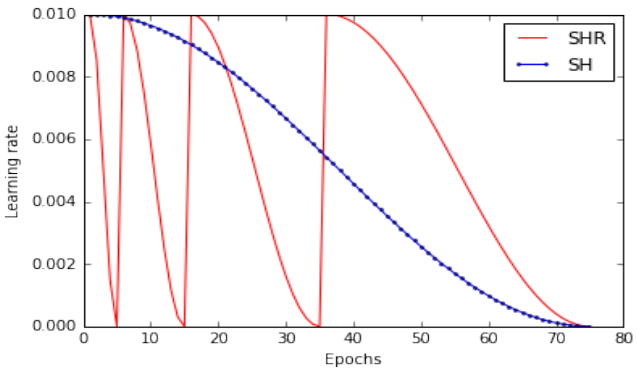

Figure 1. Different learning rate schedules for successive halving models in one evolutionary step for SH and SHR. The schedule used by SH decays once through the search, while the one used by SHR decays at the end of each successive halving step.

Later, kernels for pointwise convolutional layer are initialized as below. The kernels are applied across output of depthwise layer as standard convolution operations. The result is the same as input to the depthwise and input to the pointwise layer.

$$\begin{bmatrix} [[1] \; [0] \; [0]] \\ [[0] \; [1] \; [0]] \\ [[0] \; [0] \; [1]] \end{bmatrix}$$

After this, we can create a new building block as $SepConv - BatchNorm - ReLu$. Since this layer is two consecutive convolutional layers, we can also use it with other network operators in accordance after some small adaptations. At the end, network operators such as altering number of channels of a depthwise separable convolutional layer, splitting a depthwise separable convolutional layer up into two parallel $SepConv - BatchNorm - ReLu$ blocks, and merging operations can be used.

This operator is only used in in experiments for SHRSep (Section 4.2.4) as a sixth network operator alongside the 5 network operators which are explained in previous sections.

3.3. Child Network Selection

After mutation of a parent network is completed, and a new population of children networks are generated we use two methods for selecting the most promising child network. In this section we explain iterative Hill climbing and Successive Halving (Karnin et al., 2013) methods.

Algorithm 1 Successive Halving

Input: budget b, n children
steps = $\log_2 n$
for $i = 1$ **to** steps **do**
  $epochs$ = $\frac{b}{steps \cdot n}$
  for child **in** n **do**
    child.train($epochs$)
    child.evaluate($validation\_set$)
  end for
  Discard worst performing $n/2$ children
  $n = n/2$
end for
Output: the best performing child

3.3.1. Hill Climbing

Hill Climbing is an iterative method, which picks a model from a local neighborhood to improve the overall result. In our work, this corresponds to mutating a parent network and picking the most promising child network from the population with respect to validation accuracy, which is to be mutated in the next evolutionary step. An evolutionary step with hill climbing is explained in Figure 2. This method is based on Elsken et al. (2017).

3.3.2. Successive Halving

Successive Halving algorithm (Karnin et al., 2013) is a simple yet powerful way of maximizing the probability of correct alternative among others, given a budget which is to be distributed equally to successive halving steps. The method abandons the worst performing half of existing alternatives in each step under the budget, such that only one alternative remains at the end of the process.

In our work, we generate n children by mutating a parent network to be used in Algorithm 1. The output network of the algorithm is trained extra for $\frac{b}{steps}$ epochs, and becomes the parent network for the next evolutionary step. A graphical explanation of this process can be seen in Figure 3.

4. Experiments

In this section, we explain the experiments that we prepare to test our methods performance. We conduct and evaluate our experiments on CIFAR-10 (Krizhevsky et al.), using 10% of the training set as our validation set.

4.1. Vanilla Model

As a starting point of neural architecture search, we need a vanilla model; which we can apply our network operators. Our vanilla model is consist of 3 $Conv - BatchNorm - ReLu$, $MaxPool2d$ layers between the blocks, and a final fully connected layer.

Having the architecture of vanilla model, we want to see effects of different data augmentation methods on it. For this reason, we conduct some experiments using random horizontal flip, normalization, mixup (Zhang et al., 2018), cutout (DeVries & Taylow, 2017), random vertical and random horizontal shift for this experiment. As shown in Table 1 adding more data augmentation to such a small and shortly trained model decreases validation accuracy. However, as we want to reproduce results from Elsken et al. (2017) as close as possible; we use the same data augmentation techniques as Elsken et al. (2017) uses, namely random horizontal flip, normalization, mixup, cutout and random horizontal and vertical flip. Finally, we train vanilla model for 20 epochs.

4.2. Models

In this section, we give more details about our experiments. All experiments are repeated to get the expected accuracy and to see their stability. For the rest of the paper, we inform that the following list of settings is relevant to all experiments:

- 8 repetition of each experiment
- Hill climbing as the global search procedure
- 8 children per mutation step
- 5 mutations per child network
- Uniform distribution of network operators
- SGDR and cosine annealing (Loshchilov & Hutter, 2017)
- 0.01 initial learning rate
- 0.9 momentum
- 0.0005 weight decay
- 160 epochs per evolutionary step
- Evaluated on validation set
- Same hardware

After the architecture search ends, the best child model is trained on train + validation set for 200 epochs with learning rate of 0.025 and its performance is evaluated on test set.

4.2.1. NASH

We reimplement NASH from Elsken et al. (2017) to compare it with our methods as our baseline, which selects the models with hill climbing for children network selection. We decay our learning rate with cosine annealing, and restart it at each evolutionary step, in which all children networks are mutated and trained for $\#epochs/\#children$.

Figure 2. Example of an evolutionary step using hill climbing model selection method. A population of children networks is generated from a parent network. All members of the population are to be trained once for a fixed number of epochs. At the end, the network with the highest validation accuracy is selected to be the parent network of the next generation.

Table 1. Experiments with data augmentation on vanilla model. Subjecting a simple network to many data augmentation methods; which its capacity cannot handle, causes lower performance. In our experimental setting random horizontal flip, normalization, mixup, cutout and random horizontal and vertical flip are used even though this setting gives the lowest validation accuracy.

DATA AUG.	1	2	3	4	5	6	7	8
FLIP	√	√	√	√	√	√	√	√
NORM.	√	√	√	√	√	√	√	√
MIXUP	×	√	×	×	√	√	×	√
CUTOUT	×	×	√	×	√	×	√	√
SHIFT	×	×	×	√	×	√	√	√
VAL. ACC.	77.59	75.87	71.75	76.19	69.15	72.64	68.93	65.99

The most promising child network is selected for the next evolutionary step based on the performance on validation set as it is shown in Figure 2. Hill climbing process is repeated until the experiment budget is exhausted. At the end, the best performing child is subjected to final training.

4.2.2. SH

In this model we use successive halving for children selection per architecture search iteration. We decay our learning rate with cosine annealing, and restart it at each neural architecture search step. In total, we do not train the models more than $\#epochs$ per evolutionary step for a fair comparison with our baseline.

In this experiment, each child is trained for a small number of epochs, then all children are sorted with respect to their performances on validation set. The best half of these children are selected for further and longer training. We repeat this process until there is only one child network left. After, this child network is subjected to a final training to exhaust our training budget; and, it becomes the parent of the next generation. A complete evolutionary step with successive halving is illustrated in Figure 3.

4.2.3. SHR

As another variation of our successive halving based method, we create a new model with only one difference compared to SH. We decay our learning rate with cosine annealing, but this time we restart it at each successive halving step. The rest of the experiment follows SH directly. The difference between the learning rate schedule is depicted in Figure 1.

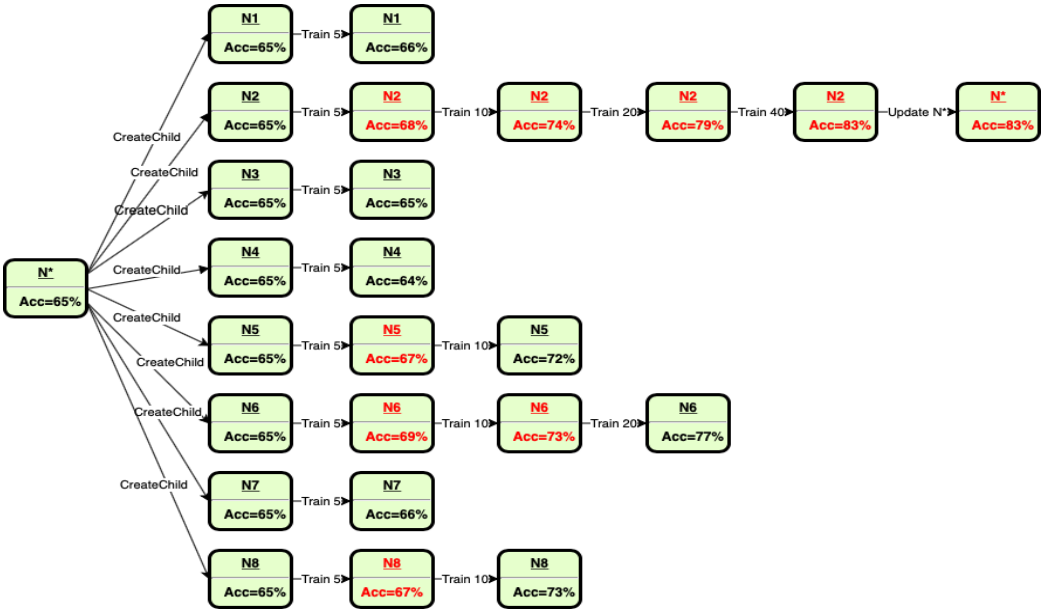

Figure 3. Example of an evolutionary step using successive halving model selection method. A population of children networks is generated from a parent network to be trained according to Algorithm 1. After each successive halving step, the worst half of the population is discarded and the best half is preserved for future steps of the algorithm. The network with the highest validation accuracy is selected to be the parent network of the next generation.

4.2.4. SHRSEP

SHRSep is another variation of SHR, with a difference in network operators. Besides using standard convolution layers, we allowed the model to have depthwise separable convolutional layers.

4.3. Experiment Budget

For our experiments, we limit the mutation process to 23 hours before the final training, and we restrict the maximum number of parameters to 20 million, which means if the number of total parameters reaches the limit, no other mutations will be applied. The number of evolutionary steps can be different for experiments as we do not put a limit for it.

5. Results

In this section we compare results from our methods and from our baseline NASH (Elsken et al., 2017). NASH has the highest accuracy ($\mu = 94.64\%$), the highest standard deviation ($\sigma = 0.38$) and the highest number of parameters in average. Our second method SHR ($\mu = 94.63\%$) reaches almost the same test accuracy as our baseline with fewer parameters and less standard deviation ($\sigma = 0.29$). SH has the least number of parameters ($\mu = 13M$) and the least standard deviation ($\sigma = 0.11$) among all methods. Its test accuracy is ($\mu = 94.57\%$). SHRSep has also a small number of parameters ($\mu = 14M$), and it has the lowest test accuracy ($\mu = 94.39\%$). The results of all experiments are summarized in Table 2.

Table 2. Experiment results from all methods. SHR and NASH have very similar results, while SHR has in average 3 Million less parameters than average NASH model. SH has the least standard deviation and number of parameters. SHRSep has the lowest test accuracy.

EXP	ACC±STD	PARAMS(M)± STD
NASH	94.64 ± 0.38	18.2 ± 6.2
SH	94.57 ± 0.11	12.9 ± 5.9
SHR	94.63 ± 0.29	14.8 ± 4.8
SHRSEP	94.39 ± 0.33	14.5 ± 4.5

Out of 8 experiments, the highest test accuracies reached by the methods are: NASH 95.07%, SH 94.70%, SHR 95.02%, SHRSep 94.98%. The best model found by NASH has 85 layers in total, with 22 $Conv - BatchNorm - ReLu$ blocks, 2 Convex Combination, 3 Concatenation and 8 Split Up layers. The best models produced by our methods have properties as follows: SH has 109 layers in total, with 25 $Conv - BatchNorm - ReLu$ blocks, 14 Convex Combination, 3 Concatenation and 11 Split Up layers; SHR has 103 layers in total, with 24 $Conv - BatchNorm - ReLu$ blocks, 7 Convex Combination, 6 Concatenation and 10

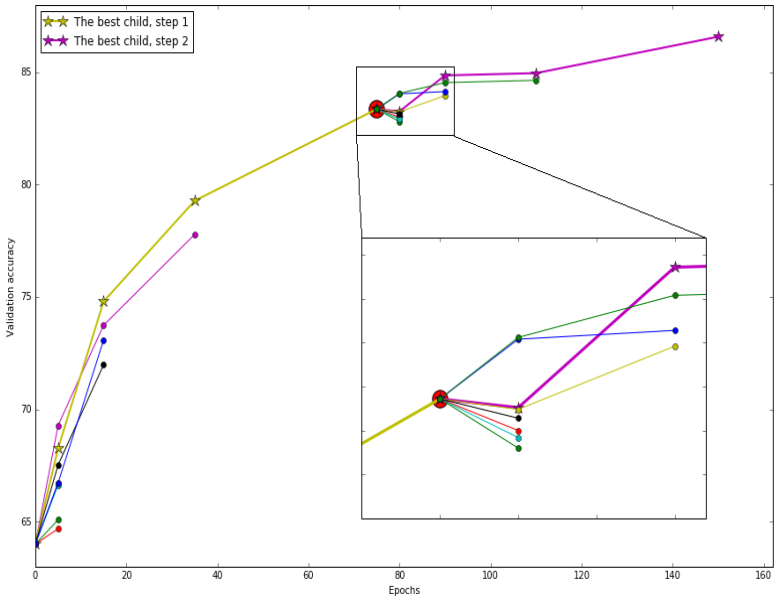

Figure 4. A snapshot from one of the experiments with successive halving, which shows two steps of neural architecture search. Firstly we generate 8 children, and follow Algorithm 1 until only one model (yellow line with star) left in the population at epoch 75 (red dot). The winner child network becomes the parent for the next evolutionary step, and 8 new children are generated from this parent. The same procedure is repeated, in which the winner is depicted with purple line with stars.

Table 3. Layer statistics from averaged experiment results with standard deviation, showing the percentage of layer types in a winner network. $MaxPool2d$ and dense layer's contributions are excluded.

EXPERIMENT	CONV BLOCKS	CONVEX COMB	CONCATENATION	SPLIT UP CONV	SEPARABLE CONV BLOCKS
NASH	$74 \pm 4\%$	$5 \pm 3\%$	$4 \pm 1\%$	$10 \pm 2\%$	0%
SH	$67 \pm 2\%$	$11 \pm 2\%$	$5 \pm 1\%$	$9 \pm 1\%$	0%
SHR	$70 \pm 2\%$	$8 \pm 4\%$	$6 \pm 2\%$	$8 \pm 2\%$	0%
SHRSEP	$52 \pm 11\%$	$8 \pm 4\%$	$4 \pm 1\%$	$10 \pm 4\%$	$23 \pm 11\%$

Split Up layers; finally SHRSep has 108 layers in total, with 23 $Conv - BatchNorm - ReLu$ blocks, 3 $SepConv - BatchNorm - ReLu$ blocks, 2 Convex Combination, 6 Concatenation and 13 Split Up layers.

It is surprising that the best SHRSep model has only 3 $SepConv - BatchNorm - ReLu$ blocks. The highest number of $SepConv - BatchNorm - ReLu$ blocks produced by a SHRSep model is 16. This model has 106 layers in total, with 11 $Conv - BatchNorm - ReLu$ blocks, 8 Convex Combination, 4 Concatenation and 6 Split Up layers, and it reached 94.62% accuracy.

Meanwhile, the lowest test accuracies are as follows: NASH 94.00%, SH 94.35%, SHR 94.14%, SHRSep 94.00%. Analysis of best and worst models and layer statistics of all models (Table 3) do not allow us to infer a correlation between a model's success and proportion of different layer types it has.

In experiments the network operators are drawn from uniform distribution, and they are applied to proper random locations. Hence, the search process is random rather than a decision making process. Our results show that even in random setting, $Conv - BatchNorm - ReLu$ blocks are favored the most by all methods. The reason for this result is that vanilla model's capacity is low, and we need convolutional layers to have enough parameters for a good representation of the data.

Obviously, as we repeat experiments for 8 times, these statistics are not very reliable. However, as the winning models have convolutional blocks the most, we can deduce that the blocks are an important factor in models' performance. It is very likely that uniform is not the best distribution for choosing network operators. For future work, one can change the distribution of network operators, such that increasing the probability of inserting blocks, or a probability distribution

can be learned.

Another important remark is about the total number of parameters. As it can be seen on Table 2, all models; which use successive halving, have fewer parameters than NASH. The reason behind this result is the successive halving budget assigned to the first step. Before the first step of successive halving, we mutate the parent network, and generate offsprings, which have different number of newly added parameters. These parameters are initialized after a series of careful calculations to preserve the functionality of the parent network, and most of them are zero. At the first step of successive halving, we train the children networks only for 5 epochs, then pick the best half of them based on validation accuracy. In 5 epochs, models with more parameters cannot be trained enough to learn all weights well enough; hence, they are thrown out from the population in early steps because of their low validation accuracy. Meanwhile, models with less number of parameters survive as they can learn their parameters better. This is a drawback of successive halving, which can also be the reason behind why successive halving based models have less test accuracy.

We also observe that the aggressive learning rate scheduler causes a natural decrease in models' accuracies when it is restarted. An example of this phenomenon can be seen in Figure 4, in which after the first evolutionary step (marked with a red dot) a decrease in validation accuracy affects most of newly generated children networks. In most cases this drop cannot be compensated in 5 epochs (initial budget), when we make the first children network selection with successive halving.

To make robust decisions about which networks to be trained for the next step of successive halving, the initial budget should be higher. With a higher initial budget, more promising networks with bigger number of parameters can be trained enough to be competitive with simple models, and the decrease in validation accuracy caused by restarting the learning rate can be compensated.

6. Future Work and Conclusions

In this work we present an extension to Elsken et al. (2017) by making use of network morphisms and successive halving, without optimizing hyper parameters. Our results show that successive halving-based methods generate networks that are not only smaller, but also competitive with our baseline (Elsken et al., 2017). There is no significant difference in term of accuracy between successive halving based methods, and (Elsken et al., 2017). Furthermore, we conclude that adding depthwise separable convolutions to our search space do not contribute to models' performance, and the best models found by SHRSep have significantly less $SepConv - BatchNorm - ReLu$ blocks than $Conv - BatchNorm - ReLu$ blocks.

It is known that successive halving does not necessarily find the optimal result, but it can find a suboptimal one. By using a small budget, we decrease our chances to get better results. A solution to this might be using a higher budget. Moreover, if we increase the budget, we can shorten the final training.

Another possible research topic would be changing network operators' distribution or defining a metric for choosing them. We can examine the metric for selecting the next mutations, rather than drawing them from a uniform distribution. In this way, the resulting architectures could be more sophisticated. This metric can check; for instance, the layer statistics or the network size to make better decisions.

We should note again that our operators only increase the capacity of networks, which can be a drawback in architecture search. This phenomenon can cause overfitting if the child network is mutated continuously, and gets larger despite being complex enough. A representative metric could be defined for future work to be used as a stopping criteria of mutation process. Another alternative to this approach is using approximate network morphisms (Elsken et al., 2018a), which can decrease models' capacity in contrast to exact network morphisms.

One could try another set of data augmentation methods different than the one we use in this work. By only looking at vanilla model's performance, different sets of methods can be used. However, we should note that vanilla model is a simple model, and its capacity is low for a combination of all data augmentation methods that we applied. Besides looking at vanilla model's performance, other proven data augmentation methods can be preferred for longer training processes.

Even though we only examine methods for neural architecture search, an exciting field of research is optimizing hyper parameters jointly with architecture (Zela et al., 2018) using reasonable computational resources, or optimizing the hyper parameters after finding the architecture.

References

Ahmed, K., Keskar, N. S., and Socher, R. Weighted transformer network for machine translation. *https://arxiv.org/pdf/1711.02132.pdf*, 2017.

Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. *https://arxiv.org/abs/1611.02167v3*, 2017.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. Smash: One-shot model architecture search through hypernetworks. *https://arxiv.org/abs/1708.05344*, 2017.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Ef-

ficient architecture search by network transformation. *https://arxiv.org/pdf/1707.04873.pdf*, 2017.

Cai, H., Yang, J., Zhang, W., Han, S., and Yu, Y. Path-level network transformation for efficient architecture search. *https://arxiv.org/pdf/1806.02639.pdf*, 2018.

Chen, T., Goodfellow, I., and Shlens, J. Net2net: Accelerating learning via knowledge transfer. *https://arxiv.org/abs/1511.05641v4*, 2016.

Chollet, F. Xception: Deep learning with depthwise separable convolutions. *https://arxiv.org/abs/1610.02357*, 2016.

Cüreşan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. High performance neural networks for visual object classification. *https://arxiv.org/pdf/1102.0183.pdf*.

DeVries, T. and Taylow, G. W. Improved regularization of convolutional neural networks with cutout. *https://arxiv.org/abs/1708.04552v2*, 2017.

Elsken, T., Metzen, J. H., and Hutter, F. Simple and efficient architecture search for convolutional neural networks. *https://arxiv.org/pdf/1711.04528.pdf*, 2017.

Elsken, T., Metzen, J. H., and Hutter, F. Efficient multi-objective neural architecture search via lamarckian evolution. *https://arxiv.org/abs/1804.09081v2*, 2018a.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search. In Hutter et al. (2018), pp. 69–86. In press, available at http://automl.org/book.

Graves, A., A.Mohamed, and Hinton, G. Speech recognition with deep recurrent neural networks. *https://arxiv.org/abs/1303.5778*, 2013.

He, K., Gkioxari, G., Dollar, P., and Girshick, R. Mask r-cnn. *https://arxiv.org/pdf/1703.06870.pdf*.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *https://arxiv.org/abs/1512.03385*, 2015.

Hutter, F., Kotthoff, L., and Vanschoren, J. (eds.). *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

Iofe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *https://arxiv.org/abs/1502.03167*, 2015.

Jin, H., Song, Q., and Hu, X. Efficient neural architecture search with network morphism. *https://arxiv.org/abs/1806.10282v1*, 2018.

Karnin, Z., Koren, T., and Somekh, O. Almost optimal exploration in multi-armed bandits. *http://proceedings.mlr.press/v28/karnin13.pdf*, 2013.

Krizhevsky, A., Nair, V., and Hinton, H. Cifar-10 (canadian institute for advanced research). *http://www.cs.toronto.edu/ kriz/cifar.html*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems 25*, pp. 1097–1105, 2012.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *IEEE*, 1998.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. Progressive neural architecture search. *https://arxiv.org/abs/1712.00559v2*, 2018a.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *https://arxiv.org/pdf/1711.00436.pdf*, 2018b.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *https://arxiv.org/abs/1806.09055*, 2018c.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *https://arxiv.org/abs/1608.03983v5*, 2017.

Lui, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., and Berg, A. C. Ssd: Single shot multibox detector. *https://arxiv.org/abs/1512.02325v5*, 2016.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. *https://arxiv.org/abs/1802.03268v2*, 2018.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. Large scale evolution of image classifiers. *https://arxiv.org/abs/1703.01041v2*, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *https://arxiv.org/abs/1802.01548v4*, 2018.

Ren, S., He, K., Girshick, R., and Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *https://arxiv.org/abs/1506.01497v3*, 2016.

Sabour, S., Frosst, N., and Hinton, G. E. Dynamic routing between capsules. *https://arxiv.org/pdf/1710.09829.pdf*, 2017.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature*, 550: 354, October 2017.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *https://arxiv.org/abs/1409.1556*, 2015.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *MIT Press Journals*, 2002.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. *https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf*, 2000.

Szegedy, C., Lui, W., Jia, Y., Sermaner, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Robinovich, A. Going deeper with convolutions. *https://arxiv.org/abs/1409.4842*, 2014.

Wei, T., Wang, C., Rui, Y., and Chen, C. W. Network morphism. *https://arxiv.org/abs/1603.01670v2*, 2016.

Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. Breaking the softmax bottleneck: A high-rank rnn language model. *https://arxiv.org/pdf/1711.03953.pdf*, 2018.

Zela, A., Klein, A., Falkner, S., and Hutter, F. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *https://arxiv.org/abs/1807.06906*, 2018.

Zhang, H., Cisse, M., Dauphin, Y. N., and Lopez-Paz, D. mixup: Beyond empirical risk minimization. *https://arxiv.org/abs/1710.09412v2*, 2018.

Zhong, Z., Yan, J., Wu, W., Shao, J., and Liu, C. Practical block-wise neural network architecture generation. *https://arxiv.org/pdf/1708.05552.pdf*, 2018.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *https://arxiv.org/abs/1611.01578v2*, 2017.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *https://arxiv.org/abs/1707.07012v4*, 2018.