SIMPLE AND EFFICIENT ARCHITECTURE SEARCH FOR CONVOLUTIONAL NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural networks have had a lot of success in recent years for variety of tasks. However, designing a well performing neural network architecture is still an exhausting trial-and-error process done by experts. We propose a new method for automatically searching for well performing CNN architectures based on a simple hill climbing procedure, network morphisms, and an anytime performance optimization stemming from a cosine based learning rate scheduler. Our method yields competitive results, whereas the resource requirements lie - in contrast to others - in the same order of magnitude as training a single network. On CIFAR-10, our method designs and trains networks with an error rate below 6% in only 12 hours on a single GPU. After one day the error is reduced to almost 5%.

1 INTRODUCTION

Neural networks have become more and more popular over the last few years due to their success in a variety of tasks such as image recognition (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012) or machine translation (Bahdanau et al.). Designing neural networks is still done by hand in most cases, which is an exhausting, time-consuming process. Additionally, the vast amount of possibilities when designing a neural network requires an expert who can restrict the search to a subset of possibilities. For these reasons it seems natural to apply some optimization algorithm which automatically searches for well-performing neural network architectures. However, most classic optimization algorithms do not suit this problem, mainly due to two reasons: Firstly, evaluating an architecture is costly as it requires training the network, i.e., methods that rely on training a vast amount of networks are impractical for most applicants/applications. Secondly, the architecture search space is discrete (e.g., number of layer, type of layer) and conditional (e.g., the number of parameters defining a layer depends on the layer type). Thus methods that rely on, e.g., differentiability or independent parameters are not applicable.

This led to a growing interest in using evolutionary algorithms (Real et al., 2017; Suganuma et al., 2017) or reinforcement learning (Baker et al., 2016; Cai et al., 2017; Zoph & Le, 2017) for automatically designing CNN architectures. Unfortunately most proposed methods are either very costly (training for days with hundreds of GPUs) or yield non-competitive performance. In this work, we aim to dramatically reduce these computational costs while still achieving competitive performance. Our contributions are as follows:

- We randomly generate networks and evaluate their performance. In our experiments, they achieve 6%-7% test error on CIFAR-10 when trained with SGDR (Loshchilov & Hutter, 2017). Future architecture search methods should considerably decrease error rates compared to this baseline, which is not the case at the moment for some methods. We hope this will serve as a very first baseline for architecture search. This also indicates that *CIFAR-10 may be too easy* and encourages researchers to consider other data sets in the future.
- We formalize and extend the work on network morphisms (Chen et al., 2015; Wei et al., 2016; Cai et al., 2017) in order to support popular network building blocks such as skip connections or Batch Normalization.
- We show that architecture search does not have to be expensive - our method finds competitive architecture, whereas the computational costs lie in the same order of magnitude as training a single network. In more detail, the cost reduction is achieved by exploiting network

morphisms/transformations (Chen et al., 2015; Wei et al., 2016) as well as exploiting the good anytime-performance learning rate scheduler from Loshchilov & Hutter (2017). We show that these two ideas incorporated in a simple hill climbing algorithm yield similar performance to other proposed methods. On CIFAR-10, our method finds and trains CNNs with an error rate below 6 % in roughly 12 hours on a single GPU. After one day the error is reduced to almost 5 %. Found models can be combined to achieve an error of 4.7 % within two days on a single GPU. On CIFAR-100, we achieve an error below 24% in one day and get close to 20% after two days.

- Our method is easy to use and easy to extend, so it hopefully can serve as a basis for future work.

We discuss related work in Section 2. The concept of network morphisms is formalized in Section 3.1. Our method is then proposed in Section 3.2. and evaluated in Section 4. We conclude our work in Section 5.

2 Related work

Automated architecture search: Due to the success of neural networks, there has been a growing interest in automated architecture search during the last years. One popular approach is to train a reinforcement learning agent with the objective of designing well-performing convolutional neural networks (Baker et al., 2016; Zoph & Le, 2017; Cai et al., 2017). In Baker et al. (2016), a RL agent is trained to sequentially choose the type of layers (convolution, pooling, fully connected) and their parameters. The RL agent from Zoph & Le (2017) also sequentially generates a string representing the network architecture. In both works, the generated networks are then trained and their performance is evaluated on a validation set, which is then used as a reward signal to update the RL agent. In a follow-up work (Zoph et al., 2017), a RL agent learned to build cells, which are then used as building blocks for a neural network. The network itself however had a fixed architecture (given the cells). Unfortunately the training of RL agents designing architecture is quite expensive as it requires training a vast amount of neural networks: Baker et al. (2016)'s experiment on CIFAR-10 required 10 GPUs for 8-10 days. Zoph & Le (2017) did their computations on 800 GPUs. To overcome this drawback, Cai et al. (2017) proposed to apply the concept of network transformations/ network morphisms. The basic idea is to generate new networks based on already trained ones with the help of these transformation in order to avoid training all networks from scratch. Their experiments on CIFAR-10 were conducted on 5 GPUs for 2-3 days. Real et al. (2017) and Suganuma et al. (2017) utilized evolutionary algorithms to generate powerful networks iteratively from a small network. Operations like inserting a layer, modifying the parameters of a layer or adding skip connections serve as "mutations" in their framework of evolution. Whereas the computational resources spent in Real et al. (2017) were also enormous (250 GPUs, 10 days), Suganuma et al. (2017) were restricted to relatively small networks due handling a population of networks. In contrast to the previous methods where network size increases over time, Saxena & Verbeek (2016) start with training a large network. In fact, they train an ensemble of neural networks, dubbed "convolution neural fabric". As they use weight sharing across networks within their fabric, training cost scales linearly with the fabric size though the number of networks within the fabric grows exponentially. In the very end, the fabric is pruned; networks with almost just zero weights are thrown out of the fabric. The authors do not report training time. Very recently, Brock et al. (2017) used hypernetworks (Ha et al.) to generate the weights for a randomly sampled network architecture with the goal of eliminating the costly process of training a vast amount of networks. However, their approaches requires training the hyper network, which is in their case a 26 layer DenseNet (Huang et al., 2016).

Network morphism/ transformation: Network transformations were (to our knowledge) first introduced by Chen et al. (2015) in the context of transfer learning. The authors described a function preserving operation to make a network deeper (dubbed "Net2Deeper") or wider ("Net2Wider") with the goal of speeding up training and exploring network architectures. Wei et al. (2016) proposed additional operations, e.g., for handling non-idempotent activation functions or altering the kernel size and introduced the term network morphism. As mentioned above, Cai et al. (2017) used network morphisms for architecture search, though they just employ the Net2Deeper and Net2Wider operators from Chen et al. (2015) as well as altering the kernel size, i.e., they limit their search space to simple architectures without, e.g., skip connections.

3 METHODS

In what follows, we formalize the concept of network morphisms, which is afterwards incorporated in a hill climbing algorithm for architecture search.

3.1 NETWORK MORPHISM

Let $\mathcal{N}(\mathcal{X})$ denote a set of neural networks defined on $\mathcal{X} \subset \mathbb{R}^n$. A network morphism is a mapping $M : \mathcal{N}(\mathcal{X}) \times \mathbb{R}^k \to \mathcal{N}(\mathcal{X}) \times \mathbb{R}^j$ from a neural network $f^w \in \mathcal{N}(\mathcal{X})$ with parameters $w \in \mathbb{R}^k$ to another neural network $g^{\tilde{w}} \in \mathcal{N}(\mathcal{X})$ with parameters $\tilde{w} \in \mathbb{R}^j$ so that

$$f^w(x) = g^{\tilde{w}}(x) \text{ for every } x \in \mathcal{X}. \tag{1}$$

In the following we give a few examples of network morphisms and how standard operations for building neural networks (e.g., adding a convolutional layer) can be expressed as a network morphism. For this, let $f_i^{w_i}(x)$ be some part of a NN $f^w(x)$, e.g., a layer or a subnetwork.

Network morphism Type I. We replace $f_i^{w_i}$ by

$$\tilde{f}_i^{\tilde{w}_i}(x) = A f_i^{w_i}(x) + b, \tag{2}$$

with $\tilde{w}_i = (w_i, A, b)$[1] . Equation (1) obviously 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

$$\tilde{f}_i^{\tilde{w}_i}(x) = C(A f_i^{w_i}(x) + b) + d, \tag{3}$$

with $\tilde{w}_i = (w_i, C, d)$. A, b are fixed, non-learnable. In this case network morphism Equation (1) 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 statistic and C, d the learnable scaling and shifting.

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

$$\tilde{f}_i^{\tilde{w}_i}(x) = \begin{pmatrix} A & \tilde{A} \end{pmatrix} \begin{pmatrix} h^{w_h}(x) \\ \tilde{h}^{w_{\tilde{h}}}(x) \end{pmatrix} + b \tag{4}$$

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 (1) can trivially be satisfied by setting $\tilde{A} = 0$.

We think of two modifications of a NN which 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 from Chen et al. (2015)). Think of $h(x)$ as 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 be formulated as a network morphism. 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 $f_i^{w_i}$ can simply be replaced by

$$f_i^{(w_i, \tilde{w}_i)} = f_i^{\tilde{w}_i} \circ f_i^{w_i} \tag{5}$$

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

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

$$\tilde{f}_i^{\tilde{w}_i}(x) = \lambda f_i^{w_i}(x) + (1 - \lambda) h^{w_h}(x), \quad \tilde{w}_i = (w_i, \lambda, w_h) \tag{6}$$

[1] In abuse of notation.

with an arbitrary function h and Equation (1) holds if λ is initialized as 1. As an alternative, one could also use

$$\tilde{f}_i^{\tilde{w}_i}(x) = f_i^{w_i}(x) + \lambda h^{w_h}(x).$$

This morphism can be used to incorporate any function, especially any non-linearities. 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. to simplify training: If $f_i^{w_i}$ itself is a sequence of layers, $f_i^{w_i} = f_{i_n}^{w_{i_n}} \circ \cdots \circ f_{i_0}^{w_{i_0}}$, then one could choose $h(x) = x$ to realize a skip from $f_{i_0}^{w_{i_0}}$ to the layer subsequent to $f_{i_n}^{w_{i_n}}$.

Note that every combinations of the network morphisms again yields a morphism. So one could for example insert a block "Conv-BatchNorm-Relu" subsequent to a Relu layer by equations (2), (3) and (5).

3.2 Architecture Search by Network morphisms

Our proposed algorithm is a simple hill climbing strategy (Russell & Norvig, 2009): we start with a small, (possibly) pretrained network. Afterwards we apply network morphisms to this initial network to generate larger, possibly better performing one when further trained. These new networks can be seen as neighbors of the initial network in the space of network architectures. Due to the network morphism Equation (1), the new networks start at the same performance as the old one. In essence, network morphisms can be seen as a way to initialize new networks at a high performing level and we do not have to train them from scratch. They can then be further trained to exploit the additional capacity obtained by the network morphism and eventually improve. Note that in principle it could also be possible that the new network is stuck in a poor performing region, which it can never escape. We are going to investigate whether this is true in our experiment section later on. The described procedure can be repeated a few times until performance on a validation set saturates. We summarize the proposed method in Algorithm 1. In our implementation, the function $ApplyNetMorph(model, n)$ (line 15 in algorithm (1)) applies n times one of three operations, uniformly sampled:

- Make the network deeper, i.e., add a "Conv-BatchNorm-Relu" block as described in the end of Section 3.1. The position where to add the block as well as the kernel size ($\in \{3, 5\}$) are uniformly sampled. The number of channels is chosen to be equal to he number of channels of the closest preceding convolution.
- Make the network wider, i.e., increase the number of channels by using the network morphism type II. The conv layer to be widened is uniformly sampled as well as the widening factor ($\in \{2, 4\}$).
- Add a skip connection from layer i to layer j (either by concatenation or addition - uniformly sampled) by using the network morphism type II or IV, respectively. i,j are also uniformly sampled.

Note that the current best model is also considered as a neighbor, i.e. our algorithm is not forced to select a new model but rather can also keep the old one.

It is important for our method that the performance of models trained for just few epochs[2] (line 17) correlates with its performance when trained to convergence. Hence, an optimization algorithm with good anytime performance is required. We chose the cosine annealing strategy from Loshchilov & Hutter (2017), whereas the learning rate is implicitly restarted: the training in line 17 always starts with a learning rate λ_{start} which is annealed to λ_{end} after $epoch_{neigh}$ epochs. We use the same learning rate scheduler in the final training (aside from a different number of epochs).

Our algorithm could also be interpreted as a very simple evolutionary algorithm with a population size of n_{neigh}, network morphisms as mutations and Lamarckism.

[2] $epoch_{neigh}$ should be small since a lot of networks need to be trained. In fact, the total number of epochs for training in our algorithm can be computed as $epoch_{total} = epoch_{neigh} n_{neigh} n_{steps} + epoch_{final}$. In our later experiments we chose $epoch_{neigh} = 17$.

Algorithm 1 Network architecture search by hill climbing

function NASH($model_0, n_{steps}, n_{neigh}, n_{NM}, epoch_{neigh}, epoch_{final}, \lambda_{end}, \lambda_{start}$)

  # $model_0 \triangleq$ model to start with, $n_{steps} \triangleq$ number of hill climbining steps
  # $n_{neigh} \triangleq$ number of neighbours, $n_{NM} \triangleq$ number of net. morph. applied
  # $epoch_{neigh} \triangleq$ number of epochs for training every neighbour
  # $epoch_{final} \triangleq$ number of epochs for final training
  # initial LR λ_{start} is annealed to λ_{end} during SGDR training

  # set current best model to start model
  $model_{best} \leftarrow model_0$
  # start hill climbing
  for $i \leftarrow 1, \ldots, n_{steps}$ **do**
    #get n_{neigh} neighbors of $model_0$ by applying n_{NM} network morphisms to $model_{best}$
    for $j \leftarrow 1, \ldots, n_{neigh} - 1$ **do**
      $model_j \leftarrow ApplyNetMorphs(model_{best}, n_{NM})$
      # train for a few epochs on training set with SGDR
      $model_j \leftarrow$ SGDRtraining$(model_j, epoch_{neigh}, \lambda_{start}, \lambda_{end})$
    end for
    # in fact, last neighbor is always just the current best
    $model_{n_{neigh}} \leftarrow$ SGDRtraining$(model_{best}, epoch_{neigh}, \lambda_{start}, \lambda_{end})$
    # get best model on validation set
    $model_{best} \leftarrow \underset{j=1,\ldots,n_{neigh}}{arg\,max} \{performance_{vali}(model_j)\}$
  end for
  # train the final model on training and validation set
  $model_{best} \leftarrow$ SGDRtraining$(model_{best}, epoch_{final}, \lambda_{start}, \lambda_{end})$
  return $model_{best}$

end function

4 EXPERIMENTS

We evaluate our method on CIFAR-10 and CIFAR-100. First, we investigate whether our considerations from the previous chapter coincide with empirical results. We also check if the interplay of modifying and training networks harms their eventual performance. Finally, we compare our proposed method with other automated architecture algorithms as well as hand crafted architectures.

We use the standard data augmentation scheme for both CIFAR datasets as described by Loshchilov & Hutter (2017) in all of the following experiments. The training set (50.000 samples) is split up in training (40.000) and validation (10.000) set for the purpose of architecture search. Eventually the performance is evaluated on the test set.

All experiments where run on Nvidia Titan X (Maxwell) GPUs. Code is implemented in Keras (Chollet et al., 2015) with TensorFlow (Abadi et al., 2015) backend.

4.1 EXPERIMENTS ON CIFAR-10

4.1.1 BASELINES

Before comparing our method to others, we first run some baseline experiments to see whether our considerations from the previous chapter coincide with empirical data. First, we investigate whether the simple hill climbing strategy is able to distinguish between good and bad-performing models. For this, we run our algorithm with $n_{neigh} = 8$ and $n_{neigh} = 1$, i.e., there is no model selection and we simply generate a single network and train it. All other parameters are the same in this experiment, namely $n_{steps} = 5, n_{NM} = 5, epoch_{neigh} = 17, epoch_{final} = 100$. We choose $\lambda_{start} = 0.05, \lambda_{end} = 0.0$ as done in Loshchilov & Hutter (2017). $model_0$ was a simple conv net:

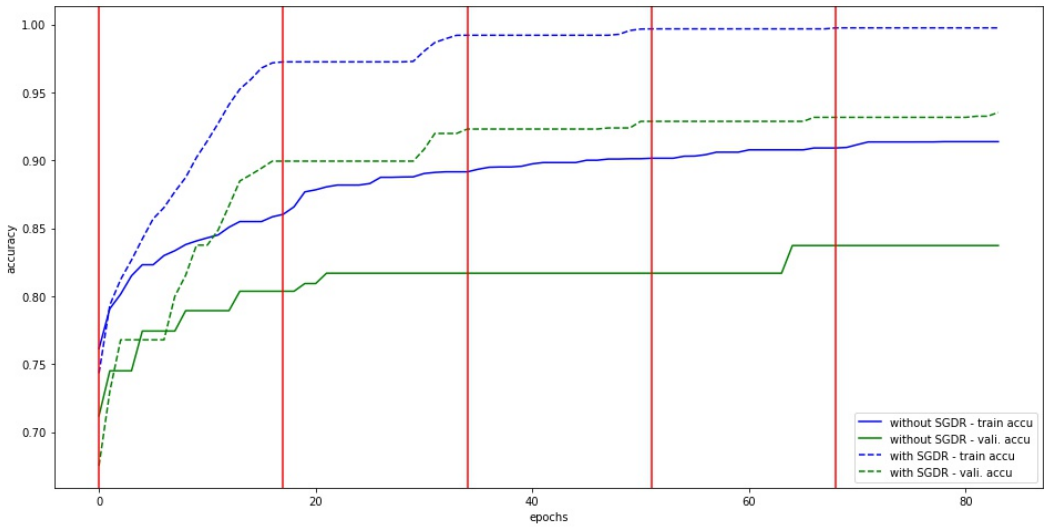

Figure 1: The best model found by Algorithm 1 tracked over time. With and without using SGDR for the training within the hill climbing (line 17). Final training (line 24) is not plotted. Red vertical lines highlight the times where network morphisms are applied (line 19).

Table 1: Baseline experiments. Runtime, # params, and error rates are averaged over 10 ($n_{neigh} = 8$) and 30 ($n_{neigh} = 1$) runs, respectively.

algorithm setting	**runtime (hrs)**	**# params (mil.)**	**error $\pm$ std. (%)**
$n_{steps} = 5, n_{neigh} = 8$	12.8	5.7	5.7 ± 0.35
$n_{steps} = 5, n_{neigh} = 8$ 10 runs	10.6	5.8	6.8 ± 0.70
$n_{steps} = 5, n_{neigh} = 1$	4.5	4.4	6.5 ± 0.76

Conv-MaxPool-Conv-MaxPool-Conv-FC-Softmax[3], which is pretrained for 20 epochs, achieving $\approx 75\%$ validation accuracy (up to 91% when trained till convergence) . If our algorithm is able to identify better networks, one would expect to get better results with the setting $n_{neigh} = 8$.
As a second experiment we turn off the cosine annealing with restarts (SGDR) during the hill climbing stage, i.e., the training in line 22 of Algorithm 1 is done with a constant learning rate, namely the starting learning rate of SGDR $\lambda_{start} = 0.05$. Note that we still use use the cosine decay for the final training.
The results for these two experiments are summarized in Table 1 . The hill climbing strategy actually is able to identify better performing models (5.7% vs. 6.5%). Notice how hill climbing prefers larger models (5.7 million parameters on average vs. 4.4 million). Also, SGDR plays an important role. The resulting models chosen by our algorithm when training is done with a constant learning rate perform similarly to the models without any model selection strategy (6.8% and 6.5%, respectively) , which indicates that the performance after a few epochs on the validation set when trained without SGDR correlates less with the final performance on the test set as it is the case for training with SGDR. With the constant learning rate, the few epochs spent are not sufficient to improve the performance of the model. Figure 1 shows the progress while running our algorithm for two (randomly picked) examples - one when trained with and one when trained without SGDR.

4.1.2 RETRAINING FROM SCRATCH.

In the this experiment we investigate whether the "weight inheritance" due to the network morphisms used in our algorithm harms the final performance of the final model. This weight inheritance can be seen as a strong prior on the weights and one could suspect that the new, larger model may not be

[3]By Conv we actually mean Conv-BatchNorm-Relu.

Table 2: Retraining generated models from scratch ($n_{steps} = 5, n_{neigh} = 8$ for all models)

	our algorithm		**retraining from scratch**	
	error (%)	runtime (hrs)	error (%)	runtime (hrs)
model 1	6.0	10.5	6.2	3.0
model 2	5.5	14.4	5.4	5.9
model 3	5.7	10.0	8.3	3.6
model 4	5.8	20.1	6.2	11.0
model 5	5.5	14.9	5.3	4.7

able to overcome a possibly poor prior. Additionally we were interested in measuring the overhead of the architecture search process, so we compared the times for generating and training a model with the time needed when training the final model from scratch. The retraining from scratch is done for the same number of epochs as the total number of epochs spent to train the model returned by our algorithm[4] . We list our findings in Table 2. The only major difference can be observed for model 3. In all other cases, the difference is below 0.5% - sometimes in favor of our algorithm, sometimes in favor of the retraining from scratch. This experiments indicates that our algorithm does not harm the final performance of a model.

Regarding the runtime, the overhead for first having to search the architecture is roughly a factor 3. We think this is a big advantage of our method and shows that architecture search can be done in the same order of magnitude as a single training.

4.1.3 Comparison to hand crafted and other automatically generated architectures

We now compare our algorithm against the popular wide residual networks (Zagoruyko & Komodakis, 2016), the state of the art model from Gastaldi as well as other automated architecture search methods. Beside our results for $n_{steps} = 5$ from the previous section, we also tried $n_{steps} = 8$ to generate larger models.
For fine tuning the results, we take snapshots of the best models from every iteration while running our algorithm following the idea of Huang et al. when using SGDR (Loshchilov & Hutter, 2017) for training. However different from Huang et al., we do not immediately get fully trained models for free, as our snapshots are not yet trained on the validation set but rather only on the training set. Hence we spent some additional resources and train the snapshots on both sets. Afterwards the ensemble model is build by combining the snapshot models with uniform weights. Lastly, we also build an ensemble from the models returned by our algorithm across all runs. Results are listed in Table 3.

The proposed method is able to generate competitive network architectures in only 12 hours. By spending another 12 hours, it outperforms most automated architecture search methods although all of them require (partially far) more time/GPUs. We do not reach the performance of the two handcrafted architectures as well as the ones found by Zoph & Le (2017) and Brock et al. (2017). However note that Zoph & Le (2017) spent by far more resources than we did. We think our trade-off between resources spent and performance is convincing.
Unsurprisingly, the snapshot models perform better. It is a simple and cheap way to improve results which everyone can consider when the number of parameters is not relevant. s

4.2 Experiments on CIFAR-100

We repeat the previous experiment on CIFAR-100; hyperparameters were not changed. The results are listed in Table 4. Unfortunately most automated architecture methods did not consider CIFAR-100.

[4]With the setting $n_{steps} = 5, epoch_{neigh} = 17, epoch_{final} = 100$ and pretraining the starting network for 20 epochs, the models that are returned by our algorithm are trained for a total number of of $20 + 17 \cdot 5 + 100 = 205$ epochs.

Table 3: Performance on CIFAR-10. For our methods the stated resources, # parameters and errors are averaged over all runs. "Resources spent" denotes training costs in case of the handcrafted models.

model	**resources spent**	**# params (mil.)**	**error (%)**
Shake-Shake (Gastaldi)	2 days, 2 GPUs	26	2.9
WRN 28-10 (Loshchilov & Hutter, 2017)	1 day, 1 GPU	36.5	3.86
Baker et al. (2016)	8-10 days, 10 GPUs	11	6.9
Cai et al. (2017)	3 days, 5 GPUs	19.7	5.7
Zoph & Le (2017)	800 GPUs, ? days	37.5	3.65
Real et al. (2017)	250 GPUs, 10 days	5.4	5.4
Saxena & Verbeek (2016)	?	21	7.4
Brock et al. (2017)	3 days, 1 GPU	16.0	4.0
Ours ($n_{steps} = 5, n_{neigh} = 8$, 10 runs)	0.5 days, 1 GPU	5.7	5.7
Ours ($n_{steps} = 8, n_{neigh} = 8$, 4 runs)	1 day, 1 GPU	19.7	5.2
Ours (snapshot ensemble, 4 runs)	2 days, 1 GPU	57.8	4.7
Ours (ensemble across runs)	1 day, 4 GPUs	88	4.4

Table 4: Performance on CIFAR-100. For our methods the stated resources, # parameters and errors are averaged over all runs. "Resources spent" denotes training costs in case of the handcrafted models.

model	**resources spent**	**# params (mil.)**	**error (%)**
Shake-Shake (Gastaldi)	7 days, 2 GPUs	34.4	15.9
WRN 28-10 (Loshchilov & Hutter, 2017)	1 day, 1 GPU	36.5	19.6
Real et al. (2017)	250 GPUs, ? days	40.4	23.7
Brock et al. (2017)	3 days, 1 GPU	16.0	20.6
Ours ($n_{steps} = 8, n_{neigh} = 8$, 5 runs)	1 day, 1 GPU	22.3	23.4
Ours (snapshot ensemble, 5 runs)	2 days, 1 GPU	73.3	20.9
Ours (ensemble across runs)	1 day, 5 GPU	111.5	19.6

Our method is on a par with Real et al. (2017) after one day on a single GPU. The snapshot ensemble performs similar to Brock et al. (2017) and an ensemble model build from the 5 runs can compete with the hand crafted WRN 28-10. The performance of the Shake-Shake network (Gastaldi) is again not reached.

5 Conclusion

We proposed a simple and fast method for automated architecture search based on a simple hill climbing strategy, network morphisms, and training via SGDR. Experiments on CIFAR-10 as well as CIFAR-100 showed that our method yields competitive results while requiring considerably less computational resources than most alternative approaches. Our algorithm is easily extendable, e.g., by other network morphisms, evolutionary approaches for generating new models, other methods for performance evaluation, e.g., learning curve prediction (Klein et al., 2017), or better resource handling strategies, e.g., hyperband (Li et al., 2016)). In this sense, we hope that our algorithm can serve as a basis for more sophisticated methods.

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 2017*, 2016.

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. Reinforcement learning for architecture search by network transformation. 2017.

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

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

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

David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. *ICLR 2017*.

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

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.

Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get M for free. *ICLR 2017*.

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

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 F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *arXiv preprint*, 2016.

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

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

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Pearson, 3 edition, December 2009. ISBN 0136042597.

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

Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. *arXiv preprint*, 2017.

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

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

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. 2017.

A SOME MODELS

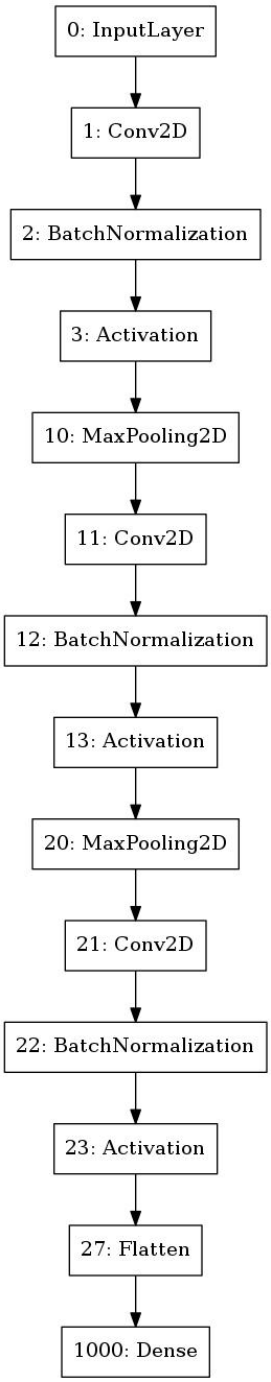

Figure 2: Initial network for our algorithm.

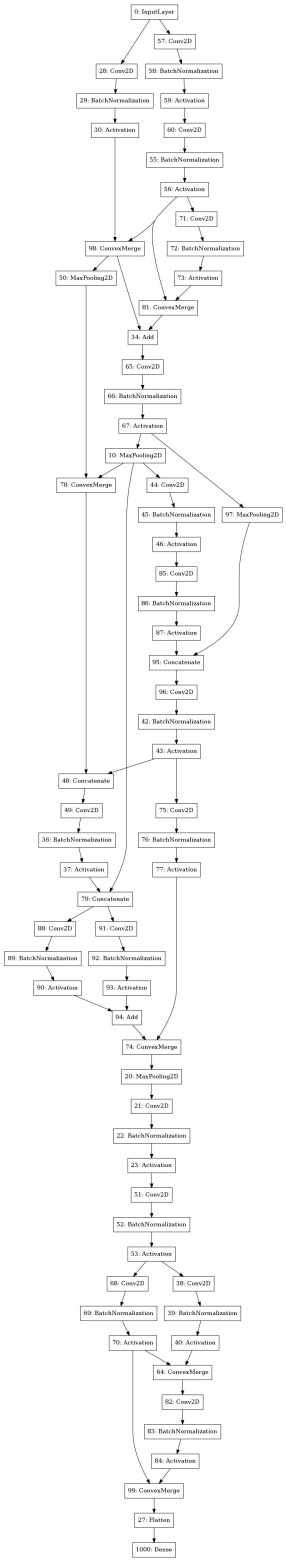

Figure 3: Network generated by our algorithm with $n_{steps} = 5$.

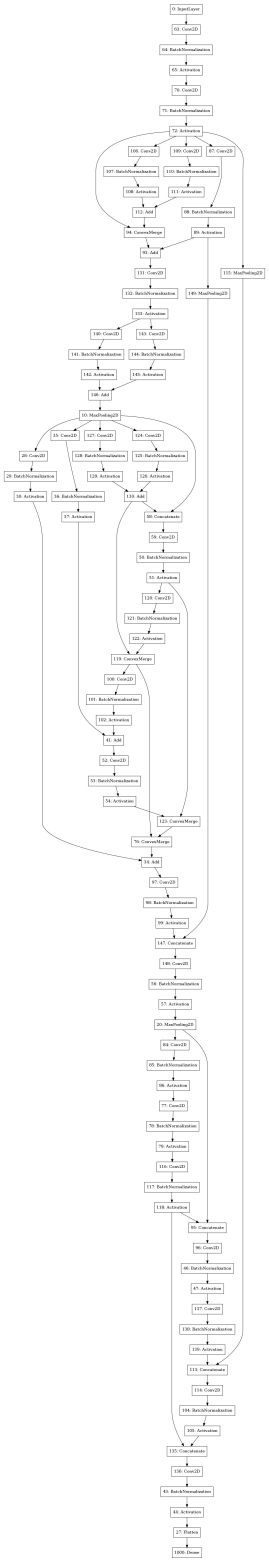

Figure 4: Network generated by our algorithm with $n_{steps} = 8$.