Deep Learning Architecture Search by Neuro-Cell-based Evolution with Function-Preserving Mutations

Martin Wistuba

IBM Research
martin.wistuba@ibm.com

Abstract. The design of convolutional neural network architectures for a new image data set is a laborious and computational expensive task which requires expert knowledge. We propose a novel neuro-evolutionary technique to solve this problem without human interference. Our method assumes that a convolutional neural network architecture is a sequence of neuro-cells and keeps mutating them using function-preserving operations. This novel combination of approaches has several advantages. We define the network architecture by a sequence of repeating neuro-cells which reduces the search space complexity. Furthermore, these cells are possibly transferable and can be used in order to arbitrarily extend the complexity of the network. Mutations based on function-preserving operations guarantee better parameter initialization than random initialization such that less training time is required per network architecture. Our proposed method finds within 12 GPU hours neural network architectures that can achieve a classification error of about 4% and 24% with only 5.5 and 6.5 million parameters on CIFAR-10 and CIFAR-100, respectively. In comparison to competitor approaches, our method provides similar competitive results but requires orders of magnitudes less search time and in many cases less network parameters.

Keywords: Automated Machine Learning · Neural Architecture Search · Evolutionary Algorithms.

1 Introduction

Deep learning techniques have been the key to major improvements in machine learning in various domains such as image and speech recognition and machine translation. Besides more affordable computational power, the proposal of new kinds of architectures such as ResNet [8] and DenseNet [9] helped to increase the accuracy. However, the selection on which architecture to choose and how to wire different layers for a particular data set is not trivial and demands domain expertise and time from the human practitioner.

Within the last one or two years we observed an increase in research efforts by the machine learning community in order to automate the search for neural network architectures. Researchers showed that both, reinforcement learning [31]

and neuro-evolution [20], are capable of finding network architectures that are competitive to the state-of-the-art. Since these methods still require GPU years until this performance is reached, further work has been proposed to significantly decrease the run time [1, 3, 15, 30, 32].

In this paper, we want to present a simple evolutionary algorithm which reduces the search time to just hours. This is an important step since now, similar to hyperparameter optimization for other machine learning models, optimizing the network architecture becomes affordable for everyone. Our presented approach starts from a very simple network template which contains a sequence of neuro-cells. These neuro-cells are architecture patterns and the optimal pattern will be automatically detected by our proposed algorithm. This algorithm assumes that the cell initially contains only a single convolutional layer and then keeps changing it by function-preserving mutations. These mutations change the structure of the architecture without changing the network's predictions. This can be considered as a special initialization such that the network requires less computational effort for training.

Our contributions in this paper are three-fold:

1. We are the first to propose an evolutionary algorithm which optimizes neuro-cells with function-preserving mutations.
2. We expand the set of function-preserving operations proposed by Chen et al. [4] to depthwise separable convolutions, kernel widening, skip connections and layers with multiple in- and outputs.
3. We provide empirical evidence that our method is outperforming many competitors within only hours of search time. We analyze our proposed method and the transferability of neuro-cells in detail.

2 Related Work

Evolutionary algorithms and reinforcement learning are currently the two state-of-the-art techniques used by neural network architectures search algorithms. With Neural Architecture Search [31], Zoph et al. demonstrated in an experiment over 28 days and with 800 GPUs that neural network architectures with performances close to state-of-the-art architectures can be found. In parallel or inspired by this work, others proposed to use reinforcement learning to detect sequential architectures [1], reduce the search space to repeating cells [30, 32] or apply function-preserving actions to accelerate the search [3].

Neuro-evolution dates back three decades. In the beginning it focused only on evolving weights [18] but it turned out to be effective to evolve the architecture as well [23]. Neuro-evolutionary algorithms gained new momentum due to the work by Real et al. [20]. In an extraordinary experiment that used 250 GPUs for almost 11 days, they showed that architectures can be found which provide similar good results as human-crafted image classification network architectures. Very recently, the idea of learning cells instead of the full network has also been adopted for evolutionary algorithms [15]. Miikkulainen et al. even propose to coevolve a set of cells and their wiring [17].

Other methods that try to optimize neural network architectures or their hyperparameters are based on model-based optimization [7, 14, 22, 26], random search [2] and Monte-Carlo Tree Search [19, 27].

3 Function-Preserving Knowledge Transfer

Chen et al. [4] proposed a family of function-preserving network manipulations in order to transfer knowledge from one network to another. Suppose a teacher network is represented by a function $f\left(\mathbf{x} \mid \boldsymbol{\theta}^{(f)}\right)$ where $\mathbf{x}$ is the input of the network and $\boldsymbol{\theta}^{(f)}$ are its parameters. Then an operation changing the network f to a student network g is called function-preserving if and only if the output for any given model remains unchanged:

$$\forall \mathbf{x}: \; f\left(\mathbf{x} \mid \boldsymbol{\theta}^{(f)}\right) = g\left(\mathbf{x} \mid \boldsymbol{\theta}^{(g)}\right) \quad . \tag{1}$$

Note that typically the number of parameters of f and g are different. We will use this approach in order to initialize our mutated network architectures. Then, the network is trained for some additional epochs with gradient-based optimization techniques. Using this initialization, the network requires only few epochs before it provides decent predictions. We briefly explain the proposed manipulations and our novel contributions to it. Please note that a fully connected layer is a special case of a convolutional layer.

3.1 Convolutions in Deep Learning

Convolutional layers are a common layer type used in neural networks for visual tasks. We denote the convolution operation between the layer input $X \in \mathbb{R}^{w \times h \times i}$ with a layer with parameters $W \in \mathbb{R}^{k_1 \times k_2 \times i \times o}$ by $X * W$. Here, i is the number of input channels, $w \times h$ the input dimension, $k_1 \times k_2$ the kernel size and o the number of output feature maps. Depthwise separable convolutions, or for short just separable convolutions, are a special kind of convolution factored into two operations. During the depthwise convolution a spatial convolution with parameters $W_d \in \mathbb{R}^{k_1 \times k_2 \times i}$ is applied for each channel separately. We denote this operation by using $\circledast$. This is in contrast to the typical convolution which is applied across all channels. In the next step the pointwise convolution, i.e. a convolution with a 1×1 kernel, traverses the feature maps which result from the first operation with parameters $W_p \in \mathbb{R}^{1 \times 1 \times i \times o}$. Comparing the normal convolution operation $X * W$ with the separable convolution $(X \circledast W_d) * W_p$, we immediately notice that in practice the former requires with $k_1 k_2 i o$ more parameters than the latter which only needs $k_1 k_2 i + i o$. Figure 1 provides a graphical representation of the network. If $X^{(l)}$ is the input for an operation in layer $l + 1$, e.g. a convolution, then we represent each channel $X^{(l)}_{:,:,i}$ by a circle. Arrows represent a spatial convolution which is parameterized by some parameters indicated by a character (in our example characters a to i). We clearly see that the depthwise convolution within the depthwise separable convolution separately operates on channels and normal convolutions operate across channels.

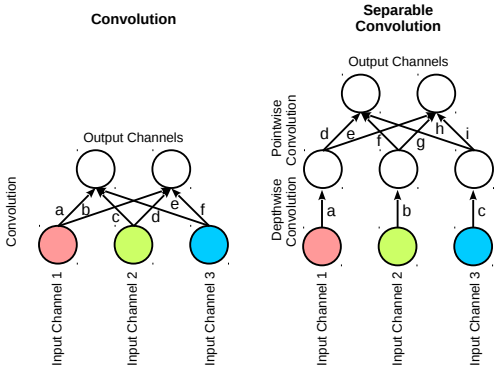


Fig. 1: Comparison of a standard convolution to a separable convolution. The separable convolution first applies a spatial convolution for each channel separately. Afterwards, a convolution with a 1×1 kernel is applied. Circles represent one channel of the feature map in the network, arrows a spatial convolution.

3.2 Layer Widening

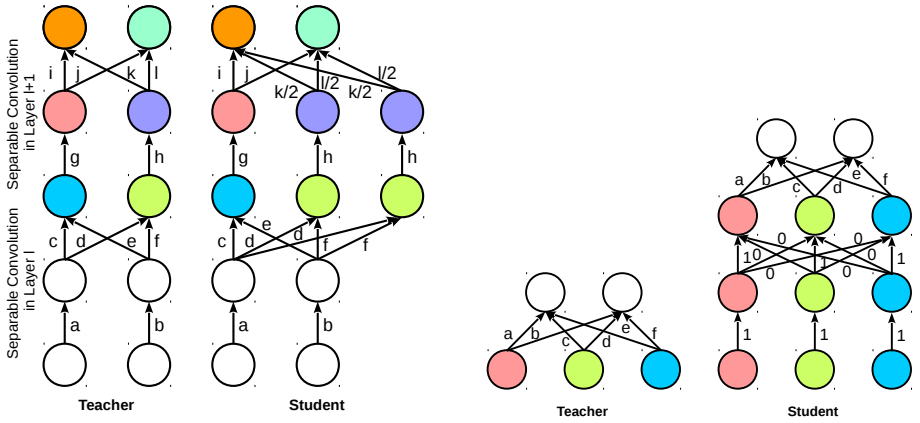
Assume the teacher network f contains a convolutional layer with a $k_1 \times k_2$ kernel which is represented by a matrix $W^{(l)} \in \mathbb{R}^{k_1 \times k_2 \times i \times o}$ where i is the number of input feature maps and o is the number of output feature maps or filters. Widening this layer means that we increase the number of filters to $o' > o$. Chen et al. [4] proposed to extend $W^{(l)}$ by replicating the parameters along the last axis at random. This means the widened layer of the student network uses the parameters

$$V^{(l)}_{\cdot,\cdot,\cdot,j} = \begin{cases} W^{(l)}_{\cdot,\cdot,\cdot,j} & j \leq o \\ W^{(l)}_{\cdot,\cdot,\cdot,r} & r \text{ uniformly sampled from } \{1,\dots,o\} \end{cases} . \tag{2}$$

In order to achieve the function-preserving property, the replication of some filters needs to be taken into account for the next layer $V^{(l+1)}$. This is achieved by dividing the parameters of $W^{(l+1)}_{\cdot,\cdot,j,\cdot}$ by the number of times the j-th filter has been replicated. If n_j is the number of times the j-th filter was replicated, the weights of the next layer for the student network are defined by

$$V^{(l+1)}_{\cdot,\cdot,j,\cdot} = \frac{1}{n_j} W^{(l+1)}_{\cdot,\cdot,j,\cdot} . \tag{3}$$

We extended this mechanism to depthwise separable convolutional layers. A depthwise separable convolutional layer at depth l is widened as visualized in Figure 2a. The pointwise convolution for the student is estimated according to Equation 2. This results into replicated output feature maps indicated by two green colored circles in the figure. The depthwise convolution is identical to the one of the teacher network, i.e. the operations with parameters a and b. Independently of whether we used a depthwise separable or normal convolution in layer l,



(a) Widening Layer l. (b) Insert a separable convolution.

Fig. 2: Visualization of different function-preserving operations. Same colored circles represent identical feature maps. Circles without filling can have any value and are not important for the visualization. Activation functions are omitted to avoid clutter.

widening it requires adaptations in a following depthwise separable convolutional layer as visualized in Figure 2a. The parameters of the depthwise convolution are replicated according to the replication of parameters in the previous layer similar to Equation 2. In our example we replicated the operation with parameters f in the previous layer. Therefore, we have now replicated spatial convolutions with parameters h. Furthermore, the parameter of the pointwise convolution (in the example parameterized by i, j, k and l) depend on the replications in the previous layers analogously to Equation 3. In our example we did not replicate the blue feature map, so the weights for this channel remain unchanged. However, we duplicated the green feature map which is transformed into the purple feature map depthwise convolution. Taking into account that this channel contributes now twice to the pointwise convolution, all corresponding weights (in the example k and l) are divided by two.

Widening the separable layer followed by another separable layer is the most complicated case. Other cases can be derived by dropping the depthwise convolutions from Figure 2a.

3.3 Layer Deepening

Chen et al. [4] proposed a way to deepen a network by inserting an additional convolutional or fully connected layer. We complete this definition by extending it to depthwise separable convolutions.

A layer can be considered to be a function which gets as an input the output of the previous layer and provides the input for the next layer. A simple function-preserving operation is to set the weights of a new layer such that the input of

the layer is equal to its output. If we assume i incoming channels and an odd kernel height and weight for the new convolutional layer, we achieve this by setting the weights of the layer with a $k_1 \times k_2$ kernel to the identity matrix:

$$V_{j,h}^{(l)} = \begin{cases} I_{i,i} & j = \frac{k_1+1}{2} \wedge h = \frac{k_2+1}{2} \\ \mathbf{0} & \text{otherwise} \end{cases} . \tag{4}$$

This operation is function-preserving and the number of filters is equal to the number of input channels. More filters can be added by layer widening, however, it is not possible to use less than i filters for the new layer. Another restriction is that this operation is only possible for activation functions σ with

$$\sigma(\mathbf{x}) = \sigma(I\sigma(\mathbf{x})) \ \forall \mathbf{x} . \tag{5}$$

The ReLU activation function $\text{ReLU}(\mathbf{x}) = \max\{\mathbf{x}, \mathbf{0}\}$ fulfills this requirement.

We extend this operation to depthwise convolutions and visualize it in Figure 2b. The parameters of the pointwise convolution V_p are initialized analogously to Equation 4 and the depthwise convolution V_d is set to one:

$$V_p = I_{i,i} \tag{6}$$

$$V_d = \mathbf{1} . \tag{7}$$

As we see in Figure 2b, this initialization ensures that both, the depthwise and pointwise convolution, just copy the input. New layers can be inserted at arbitrary positions with one exception. Under certain conditions an insertion right after the input layer is not function-preserving. For example if a ReLU activation is used, there exists no identity function for inputs with negative entries.

3.4 Kernel Widening

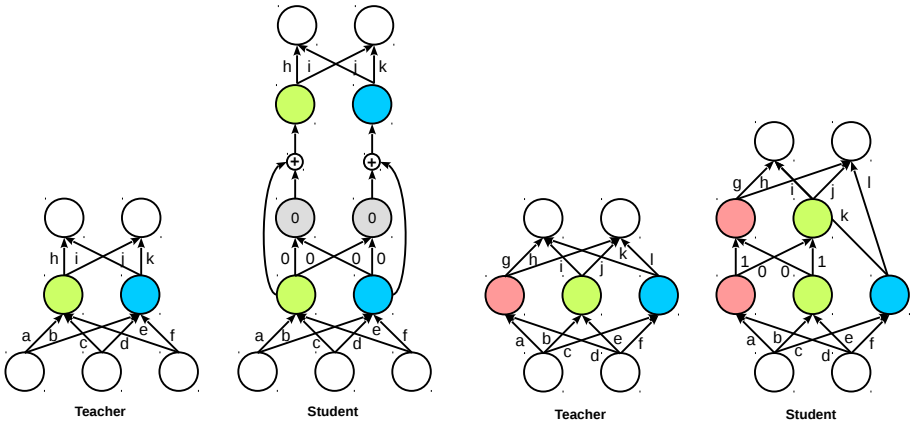
Increasing the kernel size in a convolutional layer is achieved by padding the tensor using zeros until it matches the desired size. The same idea can be applied to increase the kernel size of depthwise separable convolution by padding the depthwise convolution with zeros.

3.5 Insert Skip Connections

Many modern neural network architectures rely on skip connections [8]. The idea is to add the output of the current layer to the output of a previous. One simple example is

$$X^{(l+1)} = \sigma\left(X^{(l)} * V^{(l+1)} + X^{(l)}\right) . \tag{8}$$

Therefore, we propose a function-preserving operation which allows inserting skip connection. We propose to add layer(s) and initialize them in a way such that the output is 0 independent on the input. This allows to add a skip because now adding the output of the previous layer to zero is an identity operation. We visualized a simple example in Figure 3a based on Equation 8. A new operation is added setting its parameters to zero, $V^{(l+1)} = \mathbf{0}$, achieving a zero output. Now, adding this output to the input is an identity operation.



(a) Insert a skip with a convolution.

(b) Branch the colored layer and insert a convolution into the left branch.

Fig. 3: Visualization of different function-preserving operations. Same colored circles represent identical feature maps. Circles without filling can have any value and are not important for the visualization. Activation functions are omitted to avoid clutter.

3.6 Branch Layers

We also propose to branch layers. Given a convolutional layer $X^{(l)} * W^{(l+1)}$ it can be reformulated as

$$\text{merge}\left(X^{(l)} * V_1^{(l+1)},\ X^{(l)} * V_2^{(l+1)}\right) \quad , \tag{9}$$

where *merge* concatenates the resulting output. The student network's parameters are defined as

$$\begin{aligned} V_1^{(l+1)} &= W^{(l+1)}_{\cdot,\cdot,\cdot,1:\lfloor o/2 \rfloor} \\ V_2^{(l+1)} &= W^{(l+1)}_{\cdot,\cdot,\cdot,(\lfloor o/2 \rfloor+1):o} \quad . \end{aligned}$$

This operation is not only function-preserving, it also does not add any further parameters and in fact is the very same operation. However, combining this operation with other function-preserving operations allows to extend networks by having parallel convolutional operations or add new convolutional layers with smaller filter sizes. In Figure 3b we demonstrate how to achieve this. The colored layer is first branched and then a new convolutional layer is added to the left branch. In contrast to only adding a new layer as described in Section 3.3, the new layer has only two output channels instead of three.

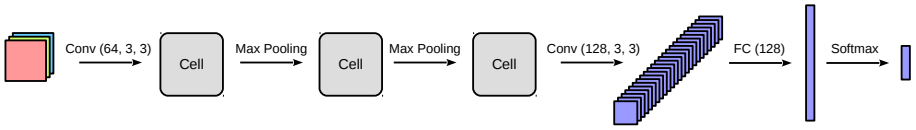


Fig. 4: Neural network template as used in our experiments.

3.7 Multiple In- or Outputs

All the presented operations are still possible for networks where a layer might have inputs from different layers or provide output for multiple outputs. In that case only the affected weights need to be adapted according to the aforementioned equations.

4 Evolution of Neuro-Cells

The very basic idea of our proposed cell-based neuro-evolution is the following. Given is a very simple neural network architecture which contains multiple neuro-cells (see Figure 4). The cells itself share their structure and the task is to find a structure that improves the overall neural network architecture for a given data set and machine learning task. In the beginning, a cell is identical to a convolutional layer and is changed during the evolutionary optimization process. Our evolutionary algorithm is using tournament selection to select an individual from the population: randomly, a fraction k of individuals is selected from the population. From this set the individual with highest fitness is selected for mutation. We define the fitness by the accuracy achieved by the individual on a hold-out data set. The mutation is selected at random which is applied to all neuro-cells such that they remain identical. The network is trained for some epochs on the training set and is then added to the population. Finally, the process starts all over again. After meeting some stopping criterion, the individual with highest fitness is returned.

4.1 Mutations

All mutations used are based on the function-preserving operations introduced in the last section. This means, a mutation does not change the fitness of an individual, however, it will increase its complexity. The advantage over creating the same network structure with randomly initialized weights is obviously that we start with a partially pretrained network. This enables us to train the network in less epochs. All mutations are applied only to the structure within a neuro-cell if not otherwise mentioned. Our neuro-evolutional algorithm considers the following mutations.

Insert Convolution A convolution is added at a random position. Its kernel size is 3×3, the number of filters is equal to its input dimension. It is randomly decided whether it is a separable convolution instead.

Branch and Insert Convolution A convolution is selected at random and branched according to Section 3.6. A new convolution is added according to the "Insert Convolution" mutation in one of the branches. For an example see Figure 3b.

Insert Skip A convolution is selected at random. Its output is added to the output of a newly added convolution (see "Insert Convolution") and is the input for the following layers. For an example see Figure 3a.

Alter Number of Filters A convolution is selected at random and widened by a factor uniformly at random sampled from $[1.2, 2]$. This mutation might also be applied to convolutions outside of a neuro-cell.

Alter Number of Units Similar to the previous one but alters the number of units of fully connected layers. This mutation is only applied outside the neuro-cells.

Alter Kernel Size Selects a convolution at random and increases its kernel size by two along each axis.

Branch Convolution Selects a convolution at random and branches it according to Section 3.6.

The motivation of selecting this set of mutations is to enable the neuro-evolutionary algorithm to discover similar architectures as proposed by human experts. Adding convolutions allows to reach popular architectures such as VGG16 [21], combinations of adding skips and convolutions allow to discover residual networks [8]. Finally the combination of branching, change of kernel sizes and addition of (separable) convolutions allows to discover architectures similar to Inception [25], Xception [5] or FractalNet [13].

The optimization is started with only a single individual. We enrich the population by starting with an initialization step which creates 15 mutated versions of the first individual. Then, individuals are selected based on the previously described tournament selection process.

5 Experiments

In the experimental section we will run our proposed method for the task of image classification on the two data sets CIFAR-10 and CIFAR-100. We conduct the following experiments. First, we analyze the performance of our neuro-evolutional approach with respect to classification error and compare it to various competitor approaches. We show that we achieve a significant search time improvement at costs of slightly larger error. Furthermore, we give insights how the evolution and the neuro-cells progress and develop during the optimization process. Additionally, we discuss the possibility of transferring detected cells to novel data sets. Finally, we compare the performance of two different random approaches in order to prove our method's benefit.

5.1 Experimental Setup

The network template used in our experiments is sketched in Figure 4. It starts with a small convolution, followed twice by a neuro-cell and a max pooling layer. Then, another neuro-cell is added, followed by a larger convolution, a fully connected layer and the final softmax layer. Each max pooling layer has a stride of two and is followed by a drop-out layer with drop-out rate 70%. The fully connected layer is followed by a drop-out layer with rate 50%. In this section, whenever we sketch or mention a convolutional layer, we actually mean a convolutional layer followed by batch normalization [11] and a ReLU activation. The neuro-cell is initialized with a single convolution with 128 filters and a kernel size of 3×3. A weight decay of 0.0001 is used.

We evaluate our method and compare it to competitor methods on CIFAR-10 and CIFAR-100 [12]. We use standard preprocessing and data augmentation. All images are preprocessed by subtracting from each channel its mean and dividing it by its standard deviation. The data augmentation involves padding the image to size 40×40 and then cropping it to dimension 32×32 as well as flipping images horizontally at random. We split the official training partitions into a partition which we use to train the networks and a hold-out partition to evaluate the fitness of the individuals.

For the neuro-evolutionary algorithm we select a tournament size equal to 15% of the population but at least two. The initial network is trained for 63 epochs, every other network is trained for 15 epochs with Nesterov momentum and a cosine learning rate schedule with initial learning rate 0.05, $T_0 = 1$ and $T_{\text{mul}} = 2$ [16]. We define the fitness of an individual by the accuracy of the corresponding network on the hold-out partition. After the search budget is exhausted, the individual with highest fitness is trained on the full training split until convergence using CutOut [6]. Finally, the error on test is reported.

5.2 Search for Networks

In Table 1 we report the mean and standard deviation of our approach across five runs and compare it to other approaches.

The first block contains several architectures proposed by human experts. DenseNet [9] is clearly the best among them, reaching an error of 4.51% with only 800 thousand parameters. Using about 25 million parameters, the error decreases to 3.42%.

The second block contains several architecture search methods based on reinforcement learning. Most of them are able to find very competitive networks but at the cost of very high search times. NASNet [32] finds the best-performing network which is on par with DenseNet but requires less parameters. However, the authors report that they required about 5.5 GPU years in order to reach this performance. Efficient Architecture Search [3] still achieves an error of 4.23% but reduces the search time drastically to ten days.

The third block contains various automated approaches based on evolutionary methods. Hierarchical Evolution [15] finds the best performing architecture

Table 1: Classification error on CIFAR-10 and CIFAR-100 including spent search time in GPU days. The first block presents the performance of state-of-the-art human-designed architectures. The second block contains results of various automated architecture search methods based on reinforcement learning. The third block contains results for automated methods based on evolutionary algorithms. The final block presents our results. For our method, we report the mean of five repetitions for the classification error and the number of parameters, the best run and the run with least network parameters.

Method	Duration	CIFAR-10		CIFAR-100	
		Error	Params	Error	Params
ResNet [8] reported by [10]	N/A	6.41	1.7M	27.22	1.7M
FractalNet [13]	N/A	5.22	38.6M	23.30	38.6M
Wide ResNet (depth = 16) [29]	N/A	4.81	11.0M	22.07	11.0M
Wide ResNet (depth = 28) [29]	N/A	4.17	36.5M	20.50	36.5M
DenseNet-BC ($k = 12$) [9]	N/A	4.51	**0.8M**	22.27	0.8M
DenseNet-BC ($k = 24$) [9]	N/A	3.62	15.3M	17.60	15.3M
DenseNet-BC ($k = 40$) [9]	N/A	**3.42**	25.6M	17.18	25.6M
NAS no stride/pooling [31]	22,400	5.50	4.2M	-	-
NAS predicting strides [31]	22,400	6.01	**2.5M**	-	-
NAS max pooling [31]	22,400	4.47	7.1M	-	-
NAS max pooling + more filters [31]	22,400	3.65	37.4M	-	-
NASNet [32]	2,000	**3.41**	3.3M	-	-
MetaQNN [1]	100	6.92	11.2M	27.14	11.2M
BlockQNN [30]	96	3.6	?	18.64	?
Efficient Architecture Search [3]	10	4.23	23.4M	-	-
Large-Scale Evolution [20]	2,600	5.4	5.4M	23.0	40.4M
Hierarchical Evolution [15]	300	**3.75**	15.7M	-	-
CGP-CNN (ResSet) [24]	27.4	6.05	**2.6M**	-	-
CoDeepNEAT [17]	?	7.30	?	-	-
Ours (mean)	0.5	4.02	5.6M	23.92	6.5M
Ours (mean)	1	3.89	7.0M	22.32	6.7M
Ours (best)	0.5	**3.57**	5.8M	22.08	6.8M
Ours (best)	1	3.58	7.2M	21.74	5.3M
Ours (least params)	0.5	4.19	**3.8M**	28.15	5.0M
Ours (least params)	1	3.77	5.8M	21.74	5.3M

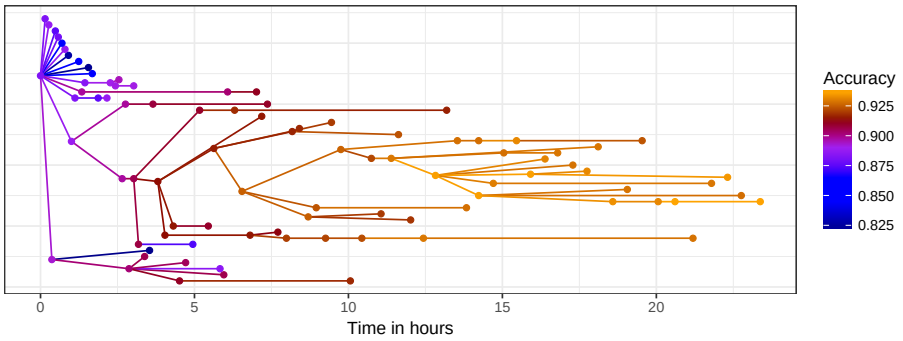


Fig. 5: Evolutionary algorithm over time. Each dot represents an individual, connections represent the ancestry. After the initialization, the algorithm quickly focuses on ancestors from only one initial individual.

among them in 300 GPU days. Methodologically, our approach also belongs into this category. We want to highlight in particular the search time required by our proposed method. Within only 12 and 24 hours, respectively, a network architecture is found which gives better predictions than most competitors and is very close to the best methods. After 12 hours of search, we report a mean classification error over five repetitions of 4.02 ± 0.376 and 23.92 ± 2.493 on CIFAR-10 and CIFAR-100, respectively. Extending the search by another 12 hours, the error reduces to 3.89 ± 0.231 and 22.32 ± 0.429.

In order to give insights into the optimization process, we visualized one run on CIFAR-10 in Figure 5 and 6. Figure 5 visualizes the fitness of each individual but also its ancestry by a phylogenetic tree [28]. The x-axis represents the time, the y-axis has no meaning. The color indicates the fitness, dots represent individuals and the ancestry is represented by edges. We notice that within the first 10 hours the fitness is increasing quickly. Afterwards, progress is slow but steady. Figure 6 provides in parallel insight which stages the final neuro-cell underwent. Over time the cell develops multiple computation branches, finally adding some skip connections. Notice, that branching the 7×7 convolution as first shown at Hour 19 has no purpose. However, this might have changed for a longer run when e.g. another layer was added in one of these branches.

5.3 Neuro-Cell Transferability

An interesting aspect is whether a neuro-cell detected on one data set can be reused in a different architecture and for a different data set. For this reason, we expanded the template from Figure 4 by duplicating the number of cells to the one shown in Figure 7. We used the cells and other hyperparameters detected in our 12 hours CIFAR-10 experiment and used the resulting networks for image classification on CIFAR-100. These models achieved an average error of 24.77% with a standard deviation of 1.61%. This result is not as good as the one achieved

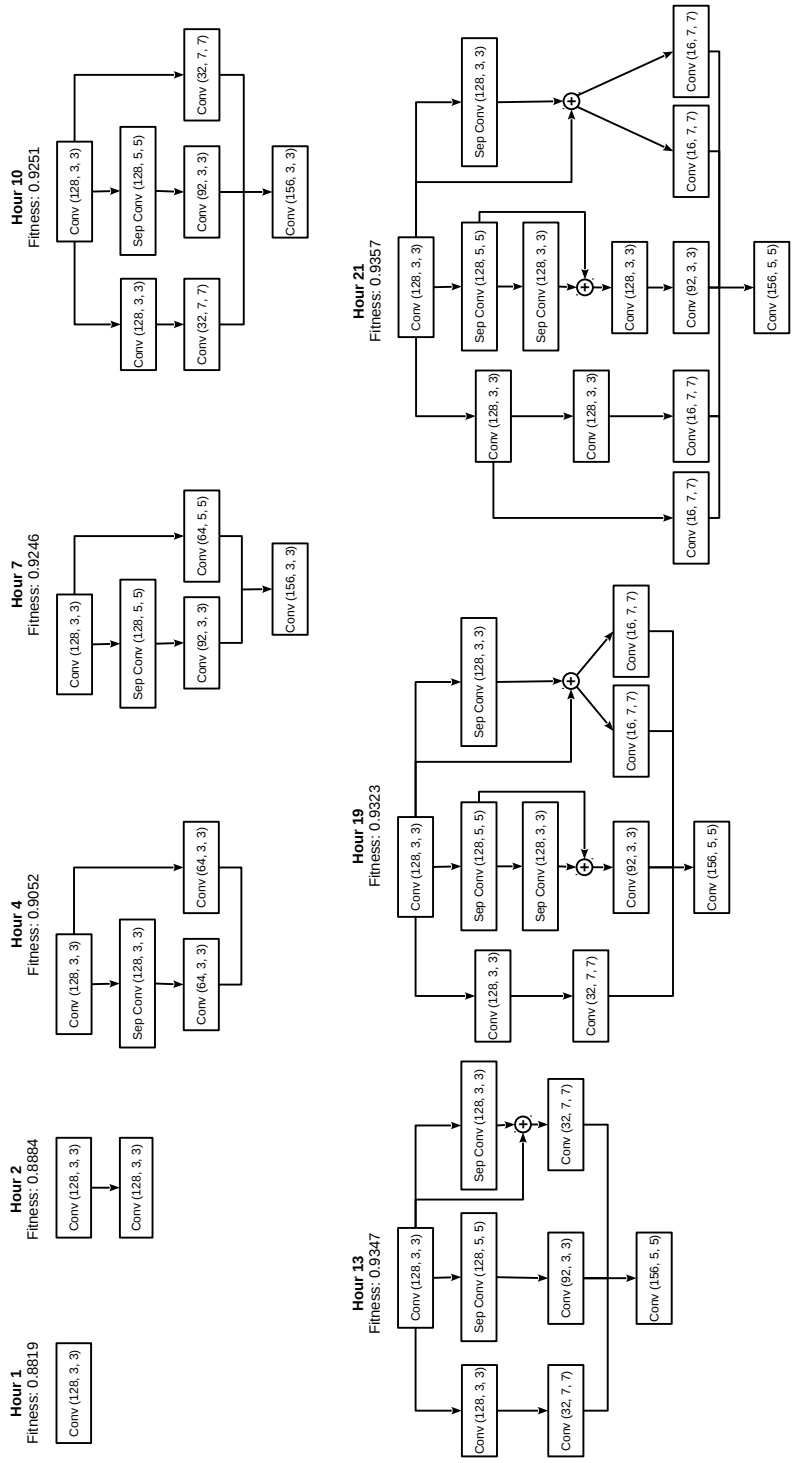


Fig. 6: Evolutionary process of the best neuro-cell found during one run on CIFAR-10. Some intermediate states are skipped.

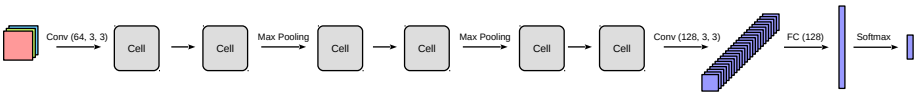


Fig. 7: Expanded template for the neuro-cell transferability experiment.

by searching for the best architecture for CIFAR-100 but therefore no new search is required for the new data set.

5.4 Random Search

In this section we will discuss the importance of our evolutionary approach by comparing it to two random network searches.

Comparison to Random Individual Selection Random individual selection is in fact not really a valid comparison because it is actually a special case of our proposed method with a tournament size of one. For this experiment, we select a random individual from the population instead of selecting the best individual of a random population subset. With this small change, we run our algorithm five times for twelve hours. We report a mean classification error of 4.55% with standard deviation 0.34%. Note, that the best of these runs achieved an error of 4.04% which is still worse than the mean error achieved when using larger tournament sizes. Thus, we can confirm that tournament selection provides better results than random selection.

Comparison to Random Mutations We conduct another experiment where we apply k mutations to the initial individual. In practice, k is dependent on the data set and not known and thus, this method is actually not really applicable. However, for this experiment, we set k to the number of mutations used for the best cell in our 12 hours experiment. In comparison to the random individual selection, this method further increases the error to 4.73% on average over five repetitions with a standard deviation of 0.63%.

6 Conclusions

We proposed a novel approach which optimizes the neural network architecture based on an evolutionary algorithm. It requires as an input a simple template containing neuro-cells, replicated architecture patterns, and automatically keeps improving this initial architecture. The mutations of our evolutionary algorithm are based on function-preserving operations which change the network's architecture without changing its prediction. This enables shorter training times in comparison to a random initialization. In comparison to the state-of-the-art, we report very competitive results and show outstanding results with respect to the

search time. Our approach is up to 50,000 times faster than some of the competitor methods with an error rate at most 0.6% higher than the best competitor on CIFAR-10.

References

1. Baker, B., Gupta, O., Naik, N., Raskar, R.: Designing neural network architectures using reinforcement learning. In: Proceedings of the International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26 (2017)
2. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**, 281–305 (2012)
3. Cai, H., Chen, T., Zhang, W., Yu, Y., Wang, J.: Reinforcement learning for architecture search by network transformation. CoRR **abs/1707.04873** (2017)
4. Chen, T., Goodfellow, I.J., Shlens, J.: Net2Net: Accelerating learning via knowledge transfer. In: Proceedings of the International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4 (2016)
5. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. CoRR **abs/1610.02357** (2016)
6. Devries, T., Taylor, G.W.: Improved regularization of convolutional neural networks with cutout. CoRR **abs/1708.04552** (2017)
7. Diaz, G.I., Fokoue-Nkoutche, A., Nannicini, G., Samulowitz, H.: An effective algorithm for hyperparameter optimization of neural networks. IBM Journal of Research and Development **61**(4), 9 (2017)
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. pp. 770–778 (2016)
9. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017. pp. 2261–2269 (2017)
10. Huang, G., Sun, Y., Liu, Z., Sedra, D., Weinberger, K.Q.: Deep networks with stochastic depth. In: Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV. pp. 646–661 (2016)
11. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. pp. 448–456 (2015)
12. Krizhevsky, A.: Learning multiple layers of features from tiny images. Tech. rep. (2009)
13. Larsson, G., Maire, M., Shakhnarovich, G.: Fractalnet: Ultra-deep neural networks without residuals. In: Proceedings of the International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26 (2017)
14. Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A.L., Huang, J., Murphy, K.: Progressive neural architecture search. CoRR **abs/1712.00559** (2017)
15. Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K.: Hierarchical representations for efficient architecture search. In: Proceedings of the International Conference on Learning Representations, ICLR 2018, Vancouver, Canada (2018)

16. Loshchilov, I., Hutter, F.: SGDR: Stochastic gradient descent with warm restarts. In: Proceedings of the International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26 (2017)
17. Miikkulainen, R., Liang, J.Z., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., Hodjat, B.: Evolving deep neural networks. CoRR **abs/1703.00548** (2017)
18. Miller, G.F., Todd, P.M., Hegde, S.U.: Designing neural networks using genetic algorithms. In: Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989. pp. 379–384 (1989)
19. Negrinho, R., Gordon, G.J.: Deeparchitect: Automatically designing and training deep architectures. CoRR **abs/1704.08792** (2017)
20. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. In: Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017. pp. 2902–2911 (2017)
21. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR **abs/1409.1556** (2014)
22. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States. pp. 2960–2968 (2012)
23. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (Jun 2002)
24. Suganuma, M., Shirakawa, S., Nagao, T.: A genetic programming approach to designing convolutional neural network architectures. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017. pp. 497–504 (2017)
25. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. pp. 1–9 (2015)
26. Wistuba, M.: Bayesian optimization combined with successive halving for neural network architecture optimization. In: Proceedings of AutoML@PKDD/ECML 2017, Skopje, Macedonia, September 22, 2017. pp. 2–11 (2017)
27. Wistuba, M.: Finding competitive network architectures within a day using UCT. CoRR **abs/1712.07420** (2017)
28. Yu, G., Smith, D.K., Zhu, H., Guan, Y., Lam, T.T.Y.: ggtree: an R package for visualization and annotation of phylogenetic trees with their covariates and other associated data. Methods Ecol. Evol. **8**(1), 28–36 (Jul 2016)
29. Zagoruyko, S., Komodakis, N.: Wide residual networks. In: Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016 (2016)
30. Zhong, Z., Yan, J., Liu, C.: Practical network blocks design with q-learning. CoRR **abs/1708.05552** (2017)
31. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. In: Proceedings of the International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26 (2017)
32. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. CoRR **abs/1707.07012** (2017)