Lamarckian Evolution of Convolutional Neural Networks

Jonas Prellberg(✉) and Oliver Kramer

University of Oldenburg, Oldenburg, Germany
{jonas.prellberg,oliver.kramer}@uni-oldenburg.de

Abstract. Convolutional neural networks belong to the most successful image classifiers, but the adaptation of their network architecture to a particular problem is computationally expensive. We show that an evolutionary algorithm saves training time during the network architecture optimization, if learned network weights are inherited over generations by Lamarckian evolution. Experiments on typical image datasets show similar or significantly better test accuracies and improved convergence speeds compared to two different baselines without weight inheritance. On CIFAR-10 and CIFAR-100 a 75 % improvement in data efficiency is observed.

Keywords: Evolutionary algorithms · Convolutional neural networks
Architecture optimization · Weight inheritance

1 Introduction

Over the last years, deep neural networks and especially convolutional neural networks (CNN) have become state-of-the-art in numerous application domains. Their performance is sensitive to the choice of hyperparameters, such as learning rate or network architecture, which makes hyperparameter optimization an important aspect of applying neural networks to new problems. However, such optimization is computationally expensive due to the lengthy training process that has to be repeated each time a new hyperparameter setting is tested. This downside applies to optimization by hand as well as to automated approaches, such as grid search, random search or evolutionary optimization.

Earlier neuroevolution works [1,5,15,16] optimize network architectures together with the network weights using an evolutionary algorithm (EA). In recent years though, EAs have mostly been applied to optimize network hyperparameters, while the training is performed with backpropagation since it offers increased efficiency for training the large and deep networks prevalent today. The usual procedure is as follows: A genotype encodes the network architecture and corresponding hyperparameters. In a genotype-phenotype process, a network is built from this description and initialized with random weights. Then, several epochs of backpropagation on a training set adjust the network weights.

A. Auger et al. (Eds.): PPSN 2018, LNCS 11102, pp. 424–435, 2018.
https://doi.org/10.1007/978-3-319-99259-4_34

Finally, the network is tested on a validation set and a metric, such as accuracy, is reported as the genotype's fitness.

Instead of randomly initializing the network weights before training, it is also possible to inherit the already learned weights of an ancestor network. This inheritance of acquired traits is a form of lamarckian evolution which, while rejected in biology, can prove useful in artificial evolution. In this work, we show that weight inheritance can drastically increase the data efficiency of an EA that optimizes neural network architectures.

The remainder of this paper is organized as follows: Sect. 2 presents related work about approaches that optimize architectures with EAs or use weight inheritance in their EAs. Section 3 describes the EA that is used in this paper and explains how the mutation with weight inheritance works. In Sect. 4 experimental results are presented and discussed. The paper ends with a conclusion in Sect. 5.

2 Related Work

Lamarckian evolution describes the idea that traits acquired over the lifetime of an individual are inherited to its offspring [14]. While rejected in biology, this approach can be beneficial for artificial evolution when there is a bi-directional mapping between genotype and phenotype. This allows to encode learned behavior back into the genotype and then apply an EA as usual. For example, Parker and Bryant [12] and Ku et al. [11] apply lamarckian evolution to neural networks by directly encoding the network weights in the genotype. This creates a simple one-to-one mapping between genotype and phenotype.

NEAT [16] is a neuroevolution algorithm that allows to grow neural networks starting with a minimal network and expanding it through mutation and principled crossover between the graphs. Because NEAT operates on single graph nodes and edges, it has been most successful on problems that can be solved with small neural networks. However, NEAT has inspired many approaches that try to extend the concept to evolving graphs of higher-level operations, such as convolutions.

Desell [3] uses a variant of NEAT to optimize the structure of a CNN that trains individual networks using backpropagation. An experiment with weight inheritance was conducted but it was not found to decrease the time necessary to train a single network to completion. Fernando et al. [4] use a microbial genetic algorithm to optimize the structure of a DPPN, which is a network that produces weights for a new network. Weight inheritance was found to improve the MSE in an image reconstruction experiment. Verbancsics and Harguess [18] test HyperNEAT [15] as a way to train CNNs for image classification. However, results were mediocre and could be substantially improved using backpropagation.

Kramer [10] uses a $(1+1)-$EA to optimize the hyperparameters defining a convolutional highway network. Suganuma et al. [17] use a modified $(1+\lambda)-$EA to optimize the structure of a CNN using a Cartesian genetic programming encoding scheme. Both approaches use small populations and only employ mutations while still achieving good results.

Weight inheritance is already employed in other works to varying degrees and for various reasons. For example, Jaderberg et al. [8] use an EA to optimize hyperparameters of static networks. It only performs mutations but inherits weights to mutated offspring. The networks are therefore trained to completion over multiple steps with potentially different hyperparameters. If one of the optimized hyperparameters is the learning rate, this effectively trains the network using a dynamic, evolved learning rate schedule. Instead, our goal is to highlight the data efficiency gains that come with weight inheritance.

Real et al. [13] use an EA with a very large population size and an unprecedented amount of computational resources to optimize the structure of a CNN. The method uses only mutation and inherits trained network weights through mutation. Training with backpropagation is performed for about 28 epochs per fitness evaluation on CIFAR-10 and CIFAR-100 and competitive results are reached. Furthermore, weight inheritance is shown to improve the test accuracy that the final network achieves. While their approach is similar to our work, we strive to keep computational demands low with the goal to reduce requirements further with the inclusion of weight inheritance.

Unrelated to evolutionary methods, the Net2Net algorithm [2] is an interesting approach to accelerate the sequential training of multiple related models. Starting from a trained model, it is possible to increase its depth or width while keeping the represented function the same. This is achieved by choosing the new weights in such a way that their effects cancel out. The training of the new model then progresses faster because the initial weights are already very good.

3 Method

To assess the influence of weight inheritance for neural network architecture optimization, design decisions regarding the optimizable hyperparameters and type of EA must be made. The goal is not to achieve state-of-the-art performance or find novel architectures but instead to show the advantages of weight inheritance. Therefore, we choose to optimize a fairly restricted architecture space which, however, is still applicable to many problems. This allows the EA to converge fast enough within our hardware resource constraints to make multiple repetitions of the same experiment feasible for statistical purposes.

The architecture search space is defined by the template presented in Fig. 1. It is made from stacked building blocks that consist of a convolutional layer followed by batch normalization [7] and a ReLU activation. The number of building blocks and the individual number of filters, kernel size and stride of the convolutional layer in each building block are subject to optimization. We statically append global average-pooling, a dense layer and a softmax activation function after the last building block, since experiments are performed specifically on image datasets.

The optimization is performed by a $(1+1)-\text{EA}$. In contrast to evolutionary algorithms with larger populations, the necessary computational resources are modest, but the method is also more prone to getting stuck in local optima in

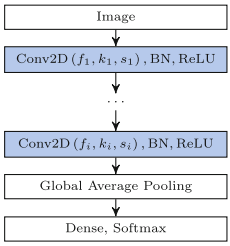

Fig. 1. Graph template defining the network architecture search space. Each building block (shown in blue) has individual hyperparameters filter count f_i, kernel size k_i and stride s_i that have to be optimized by the EA. (Color figure online)

multi-modal problems. To alleviate this, a form of niching is introduced. The evolutionary algorithm and its mutation operator are presented in more detail in the following sections.

3.1 Evolutionary Algorithm

Algorithm 1 presents the $(1+1)-$EA with niching as pseudo-code. An initial network consisting of a single convolutional layer with random filter count, random kernel size and a stride of one is created. This parent network is optimized by the EA as follows: First, a random mutation from the set of possible mutations is applied to the parent to create a child network. Next, the child network's fitness is evaluated. This means the network is trained for e epochs and the validation set accuracy is returned as its fitness. If the child's fitness is greater than the parent's fitness, the child replaces the parent.

Because this algorithm is greedy, it can get stuck in local minima. Therefore, a niching approach adapted from Kramer [10] is implemented. There is a random chance η to follow solutions that are initially worse. In such a case, a child, which has a lower fitness than its parent, is used as the parent network for a recursive call of the same algorithm. During niching, the mutate-evaluate-select-loop is repeated k times. When the last loop iteration ends, the best network found during niching is returned. If this network has a greater fitness than the original parent, it is selected. Otherwise, optimization proceeds with the original parent.

3.2 Mutation Operator

As mentioned before, the number of building blocks and the number of filters, kernel size and stride of each convolutional layer are subject to optimization. For simplicity, all these hyperparameters are chosen from predefined sets:

- Number of building blocks in $\mathbb{N}$
- Filter counts in $\mathcal{F} = \{16, 32, 64, 96, 128, 192, 256\}$
- Kernel size in $\mathcal{K} = \{1, 3, 5\}$
- Stride in $\mathcal{S} = \{1, 2\}$

```
a ← initial network
while termination condition not met do
    b ← mutate (a)
    if fitness (b) > fitness (a) then
        a ← b
    else if random() < η and not yet niching then
        c ← best network after recursion with b as initial network for k
        iterations (niching)
        if fitness (c) > fitness (a) then
            a ← c
        end
    end
end
return a
```

Algorithm 1. $(1 + 1) - \text{EA}$ with niching

Mutations are picked randomly from the list below. Each choice has a relative frequency (indicated by the multiplier in front of the list item) that determines how much more likely it is to be chosen than the mutation with a relative frequency of one. The frequencies have been chosen such that the more granular mutations, which are likely to have a smaller impact on the result, are applied less often in order to effectively use the available computation time.

- $3 \times$ *add block*: Adds a building block at a random position. The contained convolutional layer is initialized with a random filter count, random kernel size and a stride of one.
- $3 \times$ *remove block*: Removes a random building block.
- $2 \times$ *add filters*: Picks a random convolution and sets its filter count to the next greater value in $\mathcal{F}$.
- $2 \times$ *remove filters*: Picks a random convolution and sets its filter count to the next lower value in $\mathcal{F}$.
- $2 \times$ *change kernel size*: Picks a random convolution and randomly draws its kernel size from $\mathcal{K}$.
- $1 \times$ *change stride*: Picks a random convolution and randomly draws its stride from $\mathcal{S}$.

All random choices within each mutation, such as picking a random convolution or kernel size, are drawn uniformly at random from the appropriate set.

The mutation operator is forced to modify the network. A history of all previously evaluated networks is maintained and mutations are repeatedly applied to the parent network until a network is created that has not been evaluated before. Furthermore, only networks with at most three convolutions of stride two are allowed because the image inputs of CIFAR are only 32×32 and each stride-two convolution halves the side lengths.

3.3 Weight Inheritance

Each network is associated with a set of weights that contains, for example, kernels and biases for convolutional layers. When creating the initial parent network, these weights are randomly initialized in an appropriate fashion, e.g. Glorot [6] initialization. However, once a network has been evaluated its weights contain useful, learned values. When the mutation operator is applied, most of these weights are kept intact. The mutations *add block*, *remove block* and *change stride* do not influence existing weights so that all of them can be reused. The additional weights that belong to the convolutional and batch-normalization layers created by *add block* are randomly initialized. However, the mutations *add filters*, *remove filters* and *change kernel size* influence the shapes of some existing weights. For example, since the shape of a convolutional layer's kernel depends on its input and output shape, adding filters to a layer also affects the successive layer. In such cases, the affected weights are randomly reinitialized, while all other weights are reused.

4 Experiments

Training neural networks for image classification typically takes lots of resources. Hence, improving data-efficiency would be of great value. Therefore, we choose to experiment on the standard image benchmarks CIFAR-10 and CIFAR-100.

4.1 Setup

The mutation operator that employs weight inheritance is compared to a mutation operator that randomly reinitializes all network weights after each mutation. Otherwise, the same EA with the same hyperparameters is used on both datasets. The niching rate and depth are set to $\eta = 0.1$ and $k = 5$ respectively.

During each fitness evaluation, a network is trained for e epochs and subsequently its performance is assessed on the validation set. Choosing e is a trade-off between evaluation speed and the accuracy of the fitness assessment. If e is very low, evaluation is fast but networks are not trained to completion. Therefore the reported fitness will usually be lower than what the network could actually achieve given enough training time. Consequently, large but accurate networks have difficulty competing with smaller networks which train faster but might reach a lower final accuracy. If e is very high, these problems vanish but the evaluation takes a long time. Since many evaluations are necessary for large search spaces, this is impractical. Weight inheritance is supposed to offset some of the problems that come with small choices of e. The EA gets a budget of n total training epochs as its termination condition. This allows for a comparison of accuracy in terms of training examples that each experiment has processed. Also, the choices of n and e together influence how many generations, i.e. mutations, are possible within the total training epoch budget.

We propose an EA with weight inheritance and $e = 4$ training epochs per fitness evaluation. Note that 4 epochs is not sufficient to train networks that

work well on CIFAR to completion. The comparison baseline is an EA that does not use weight inheritance with two different epoch settings. The first baseline, which will be called baseline I, also uses 4 training epochs per evaluation in order to allow for a direct comparison. This allows us to show that the algorithm with weight inheritance is more data efficient and has better final accuracy all else being equal. The second baseline, which will be referred to as baseline II, uses $e = 16$ training epochs per evaluation. This significantly longer training time is more in line with the traditional approach of optimizing neural network architectures. It allows us to show that our observations still hold here and we do not simply trade a lower final accuracy for data efficiency. During evolution the best network is saved in regular intervals. After the EA finishes, these checkpoints are trained to completion and evaluated on the test set. We use this to compare test accuracies at one point during the evolution and after the evolution is finished.

The experiments are repeated 20 times with different random seeds to account for variance introduced by the randomness that is inherent to the EA and also the network training. Using the results from these repetitions, we perform significance tests using the one-sided Mann-Whitney U test. It was chosen because the sample sizes are small as each sample requires considerable time to create.

4.2 Training Details

The datasets have been split into a training, validation, and test set which contain 45k, 5k and 10k examples respectively for both CIFAR-10 and CIFAR-100. During a fitness evaluation, backpropagation is performed on the training set for e epochs and the validation set accuracy is reported as the network's fitness. All training phases are performed using a cross-entropy loss, the Adam [9] optimizer, a batch size of 512 and a learning rate of 0.001. Adam's state, i.e. first and second moment estimates, is not inherited during mutation. The test set is only used after all experiments have finished to evaluate saved network checkpoints. These checkpoints are trained to completion using a learning rate schedule: 10^{-3} until epoch 10, 10^{-4} until epoch 20 and 10^{-5} until epoch 30.

4.3 Results

Figure 2 compares three experimental settings on CIFAR-10 and CIFAR-100 with a total epoch budget of 512. The EA with weight inheritance outperforms the comparison baselines that do not use weight inheritance on both datasets. The accuracy plateau is reached more quickly and higher test accuracy is achieved.

For CIFAR-10, weight inheritance experiments reach a mean test accuracy of $85\,\% \pm 2\%$ after only 128 total training epochs. In comparison, baseline I experiments reach a mean test accuracy of $82\,\% \pm 1\%$ after 512 epochs. This means that the EA with weight inheritance achieves significantly ($p < 0.01$) higher accuracy than baseline I in 75 % less total training epochs. Baseline II experiments reach a test accuracy of $85\,\% \pm 3\%$ after 512 epochs. This is slightly, though not significantly, lower than the test accuracy of the weight inheritance experiments. After

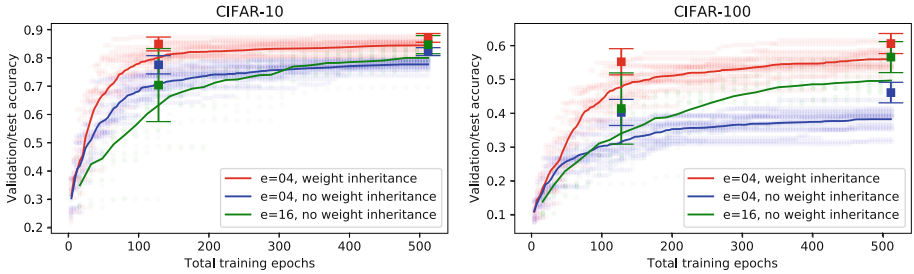

Fig. 2. Comparison of the EA with weight inheritance and $e = 4$ epochs against two baselines without weight inheritance and $e \in \{4, 16\}$ epochs on CIFAR-10 and CIFAR-100. Each dot represents the best validation accuracy achieved so far during an EA run at the respective epoch. Each line runs through the mean of the dots that are from the same experimental setting after the same amount of total epochs. Each box shows the average test accuracy after training the networks to convergence and the boxplot whiskers represent one standard deviation. (Color figure online)

512 epochs, the weight inheritance experiments reach a mean test accuracy of $87\,\% \pm 2\,\%$ which is significantly ($p < 0.01$) higher than baseline II at 512 epochs.

For CIFAR-100 results look very similar. After 128 epochs, the weight inheritance experiments achieve a mean test accuracy of $55\,\% \pm 4\,\%$. In contrast, baseline I experiments reach a significantly ($p < 0.01$) lower mean test accuracy of $46\,\% \pm 3\,\%$ after 512 epochs. Again, this is an improvement using 75 % less total training epochs. Baseline II experiments achieve a (not significantly) higher mean test accuracy of $57\,\% \pm 5\,\%$ after 512 epochs. Running the weight inheritance experiments for all 512 epochs as well results in a mean test accuracy of $61\,\% \pm 3\,\%$ which now is significantly ($p < 0.01$) higher than baseline II at 512 epochs.

In summary, weight inheritance experiments on CIFAR-10 and CIFAR-100 have shown to achieve significantly ($p < 0.01$) higher accuracy using a quarter of the total training epochs when compared to baseline I that uses the same amount of training epochs per fitness evaluation. Furthermore, final accuracy after 512 epochs is also significantly ($p < 0.01$) higher compared to baseline II experiments which benefited from more training epochs per fitness evaluation.

To get an idea how the evolutionary process modifies the genotypes, consider Fig. 3. It shows how the genome length, i.e. the number of building blocks in the corresponding networks, changes over the course of evolution. All EA runs are initialized with a genotype that contains a single building block. During the evolutionary process, increasingly larger genotypes are evaluated as they offer more accuracy than genotypes with fewer building blocks. The weight inheritance experiments and baseline II both settle around an average of 7 building blocks, whereas baseline I networks contain an average of 6 building blocks.

Table 1 lists minimum, mean and maximum test accuracies of the CIFAR experiments for specific checkpoint epochs. When weight inheritance is used, minimum, mean and maximum accuracies are higher than their baseline

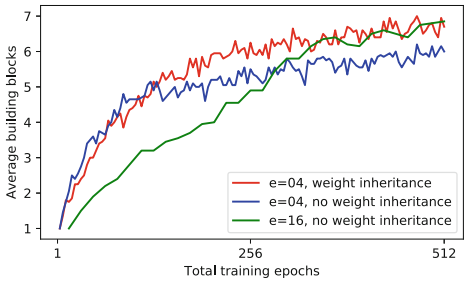

Fig. 3. Average (of all EA runs) number of building blocks in the genome during the optimization process on CIFAR-100

counterparts at all tested checkpoints. None of the results reach state-of-the-art performance, which was, as already pointed out, not the goal of this work. Our best evolved network on CIFAR-100 without data augmentation reaches a test accuracy of 66.1 % after 512 total epochs and required 6×10^{10} FLOPS[1] to find. This takes about 1.5 days on a single Nvidia K40 GPU.

Table 1. Test accuracies at the two checkpoints on CIFAR-10 and CIFAR-100

Settings			128 total epochs			512 total epochs		
Data	Inheritance	e	Min	Mean	Max	Min	Mean	Max
C10	Yes	4	**79.1**	**85.0±2.4**	**89.0**	**83.3**	**87.2±1.5**	**89.3**
	No	4	68.3	77.6±3.2	81.8	78.9	82.3±1.4	84.2
	No	16	32.5	70.4±12.6	85.5	76.6	84.8±3.1	88.9
C100	Yes	4	**47.7**	**55.3±3.8**	**61.1**	**56.1**	**60.7±2.9**	**66.1**
	No	4	31.7	40.2±3.8	46.0	39.8	46.1±3.0	52.2
	No	16	25.9	41.4±10.3	57.7	46.6	56.7±4.5	63.0

Additional experiments with 10 repetitions each have been performed on the MNIST and Fashion-MNIST datasets. The results are shown in Fig. 4. On both datasets, improvements from weight inheritance over its baselines are marginal. This is expected, as both datasets are easy to solve compared to CIFAR and can be learned quickly by small networks. Still, there is no deterioration in performance from using weight inheritance either.

[1] The FLOPS estimate for a single network is based on the FLOPS reported by the TensorFlow profiler to process a single example multiplied by 4 epochs, 98 batches per epoch and batch size 512. The total FLOPS of the EA run is the sum of the FLOPS estimates for all networks that were trained during the optimization.

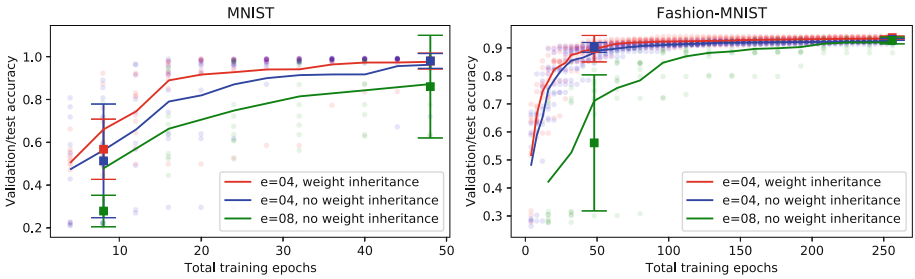

Fig. 4. Comparison of the EA with weight inheritance and $e = 4$ epochs against two baselines without weight inheritance and $e \in \{4, 8\}$ epochs on MNIST and Fashion-MNIST. See Fig. 2 for an explanation of the plot.

4.4 Discussion

The tradeoff between few and many training epochs per fitness evaluation that is explained in Sect. 4.1 has a visible effect in Fig. 2. At the beginning of each experiment, baseline I outperforms baseline II but at some point this relationship inverts. This is because small networks, which require only few epochs to reach good accuracy, are still sufficient to increase the validation set accuracy in the beginning of the experiment. However, at some point larger networks become necessary to further improve the results. These networks require more training time, making it easier for the algorithm that trains networks longer during fitness evaluation to progress. Therefore the green and blue graphs intersect. This happens earlier for CIFAR-100 because it is a harder problem than CIFAR-10.

We have seen weight inheritance experiments consistently outperform their baselines on CIFAR-10 and CIFAR-100 but could not observe a significant difference on the MNIST or Fashion-MNIST datasets. We did not find any instances of our experiments where weight inheritance was harmful, but this need not be the case generally: Just like in our work, most recent neuroevolution publications only use mutation operators and refrain from performing crossover. While this is usually motivated by the difficulty of designing a useful network crossover operator, crossover might also bring problems with regard to weight inheritance. Similarly to choosing a bad initialization before starting the training of a network, building a new network from trained parts of different networks could leave it in a region of the parameter space that is hard to optimize.

5 Conclusion

Evolutionary algorithms show promise as a way to automatically discover appropriate network architectures for new problems, but their usefulness is limited by their enormous computational requirements. Optimizing deep neural network architectures is computationally expensive because networks have to be retrained for each fitness evaluation. Therefore, approaches to lower these requirements are of great value.

We show that an evolutionary algorithm with a weight inheritance scheme generally achieves equal or higher accuracy compared to baselines that do not use weight inheritance and benefit from more training epochs per fitness evaluation. The fitness convergence speed is improved, sometimes making it possible to drastically reduce the number of total training epochs, while achieving test accuracies comparable to the baselines. Specifically, on both CIFAR-10 and CIFAR-100 weight inheritance increases data efficiency by 75 % with comparable test accuracy. The resulting speedup makes evolutionary algorithms a lot more viable for application to neural network architecture optimization even on hard problems. If accuracy is more important than training time, weight inheritance can also lead to a higher final test accuracy in some cases. Most importantly though, there has been no instance where weight inheritance was harmful. All results show either equally good or better results than the baselines. Thus it seems generally advisable to try the inclusion of weight inheritance schemes when mutation operators are used for neural network architecture optimization.

A promising research direction for future work will be to explore the interactions between weight inheritance and crossover operators. Also, further decreasing the time necessary for each training step in the evolutionary process is an important goal. For example, integrating the Net2Net [2] algorithm with the evolutionary algorithm might offer better results than randomly initializing additional new weights and allow for even less training steps.

References

1. Baluja, S.: Evolution of an artificial neural network based autonomous land vehicle controller. IEEE Trans. Syst. Man Cybern. Part B (Cybern.) **26**(3), 450–463 (1996)
2. Chen, T., Goodfellow, I., Shlens, J.: Net2net: accelerating learning via knowledge transfer. In: Proceedings of the International Conference on Learning Representations (ICLR 2016) (2016)
3. Desell, T.: Large scale evolution of convolutional neural networks using volunteer computing. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2017), pp. 127–128. ACM, New York (2017). https://doi.org/10.1145/3067695.3076002
4. Fernando, C., et al.: Convolution by evolution: differentiable pattern producing networks. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2016), pp. 109–116. ACM, New York (2016). https://doi.org/10.1145/2908812.2908890
5. Garcia-Pedrajas, N., Hervas-Martinez, C., Munoz-Perez, J.: Covnet: a cooperative coevolutionary model for evolving artificial neural networks. IEEE Trans. Neural Netw. **14**(3), 575–596 (2003). https://doi.org/10.1109/TNN.2003.810618
6. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Teh, Y.W., Titterington, M. (eds.) Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 9, pp. 249–256. PMLR, Chia Laguna Resort, Sardinia, 13–15 May 2010. http://proceedings.mlr.press/v9/glorot10a.html
7. 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, pp. 448–456 (2015)

8. Jaderberg, M., et al.: Population Based Training of Neural Networks. ArXiv e-prints, November 2017
9. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: The International Conference on Learning Representations (ICLR 2015), December 2015
10. Kramer, O.: Evolution of convolutional highway networks. In: Sim, K., Kaufmann, P. (eds.) EvoApplications 2018. LNCS, vol. 10784, pp. 395–404. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77538-8_27
11. Ku, K.W.C., Mak, M.W., Siu, W.C.: A study of the Lamarckian evolution of recurrent neural networks. IEEE Trans. Evol. Comput. **4**(1), 31–42 (2000). https://doi.org/10.1109/4235.843493
12. Parker, M., Bryant, B.D.: Lamarckian neuroevolution for visual control in the quake ii environment. In: 2009 IEEE Congress on Evolutionary Computation, pp. 2630–2637, May 2009. https://doi.org/10.1109/CEC.2009.4983272
13. Real, E., et al.: Large-scale evolution of image classifiers. In: Proceedings of the 34th International Conference on Machine Learning (ICML 2017) (2017). https://arxiv.org/abs/1703.01041
14. Sasaki, T., Tokoro, M.: Comparison between Lamarckian and Darwinian evolution on a model using neural networks and genetic algorithms. Knowl. Inf. Syst. **2**(2), 201–222 (2000). https://doi.org/10.1007/s101150050011
15. Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based encoding for evolving large-scale neural networks. Artif. Life **15**(2), 185–212 (2009). https://doi.org/10.1162/artl.2009.15.2.15202
16. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002). https://doi.org/10.1162/106365602320169811
17. 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), pp. 497–504. ACM, New York (2017). https://doi.org/10.1145/3071178.3071229
18. Verbancsics, P., Harguess, J.: Image classification using generative neuro evolution for deep learning. In: 2015 IEEE Winter Conference on Applications of Computer Vision, pp. 488–493, January 2015. https://doi.org/10.1109/WACV.2015.71