Neural Architecture Evolution in Deep Reinforcement Learning for Continuous Control

Jörg K.H. Franke[*,1], **Gregor Koehler**[*,2], **Noor Awad**[1], **Frank Hutter**[1,3]
[1]University of Freiburg
[2]German Cancer Research Center (DKFZ)
[3]Bosch Center for Artificial Intelligence

Abstract

Current Deep Reinforcement Learning algorithms still heavily rely on handcrafted neural network architectures. We propose a novel approach to automatically find strong topologies for continuous control tasks while only adding a minor overhead in terms of interactions in the environment. To achieve this, we combine Neuroevolution techniques with off-policy training and propose a novel architecture mutation operator. Experiments on five continuous control benchmarks show that the proposed Actor-Critic Neuroevolution algorithm often outperforms the strong Actor-Critic baseline and is capable of automatically finding topologies in a sample-efficient manner which would otherwise have to be found by expensive architecture search.

1 Introduction

The field of Deep Reinforcement Learning (DRL) bears a lot of potential for meta-learning. DRL has recently achieved remarkable success in areas ranging from playing games [1–3], locomotion control [4] and visual navigation [5] to robotics [6, 7]. However, all of these successes of DRL were based on manually-chosen neural architectures, rather than based on *learned* architectures.

In this paper, we introduce a novel and efficient method for learning the architecture used in DRL algorithms for continuous control *online*. To achieve this, we jointly learn the architecture of both Actor and Critic in a Q-Function Policy Gradient setting based on TD3 [8]. Specifically, our contributions are as follows:

- We combine concepts from neuroevolution and off-policy RL training to evolve and train a population of actor and critic architectures in a sample-efficient way.
- We propose a novel genetic operator based on network distillation [9] for stable architecture mutation.
- Our method is the first to adapt the neural architecture of RL algorithms *online*, based on off-policy training with the use of environment interactions from architecture evaluations shared in a global replay buffer.
- Our method finds optimal architectures but only has a small overhead in terms of steps in the environment over a RL run with a single fixed architecture.

2 Background

Our proposed approach is based on Neuroevolution, a technique to optimize neural networks including their architecture using Genetic Algorithms (GAs) [10–12]. First, a population of computation graphs with minimal complexity, represented by genomes, is created. Using genetic operators such as adding nodes or edges, additional structure is added incrementally. Different approaches in Neuroevolution

[*]Equal contribution. Correspondence to frankej@cs.uni-freiburg.de and g.koehler@dkfz.de.

usually differ in what is represented by the nodes and edges. In [10, 11], each node represents an individual neuron and the edges determine the inputs to each neuron. Recent work extending Neuroevolution for larger network architectures used nodes to represent whole layers in a network, encoding the layer using its type-specific hyperparameters (e.g. kernel size in convolutional layers) [13]. In this paper we follow a similar approach, encoding the network architecture with multi layer perceptrons (MLPs). In contrast to the few existing works for learning neural architectures for RL (using blackbox evolutionary optimization [14, 15] or multi-fidelity Bayesian optimization [16]), our approach optimizes the neural network architecture online, substantially improving sample-efficiency.

3 Methods

The foundation of Actor Critic Neuroevolution (ACN) is a genetic algorithm which evolves a population of $\mathcal{N}$ agents $\mathcal{P} = \{\mathcal{A}_1, \ldots, \mathcal{A}_\mathcal{N}\}$. We associate each agent $\mathcal{A}_n = [f_n, \psi_n^a, \theta_n^a, \psi_n^c, \theta_n^c]$ with a fitness value f_n, along with topology descriptions ψ_n^a and ψ_n^c, for actor and critic respectively, as well as their parameters θ_n^a and θ_n^c.

For simplicity and comparability, we restrict the topology to standard MLPs for both actor and critic. After initializing the networks of each individual, we evaluate the actor MLP in the environment to obtain initial fitness values. With the initial fitness values in place, the evolution loop runs for $\mathcal{G}$ generations utilizing tournament selection [17] for actor and critic individually to find the candidates for the next generation. Since mutation changes the actor, the critic can not be conditioned on the actor's behaviour and needs to be generally optimal. In the following sub-sections, we first explain the components of our algorithm and then how they are integrated in a single algorithm.

3.1 Distilled Topology Mutation

We introduce a novel mutation operator that acts on the topology of both the actor and the critic networks of the population. In order to mutate the actor and critic of an individual in a stable way, our proposed method operates in two stages. First, we jointly grow the topology of both networks in order to increase their capacity. With probability p_L, this growing mechanism appends another hidden layer of the same size as the previous last hidden layer to the respective networks; and with probability $1 - p_L$, an existing hidden layer is chosen at random and a random number of new nodes are added to this layer. Both types of changes in topology are applied identically for actor and critic networks.

As the necessary initialization of the additional parameters introduced by growing the topology also changes both the critic's estimate of the state-action values, as well as the policy of the actor, we propose a second stage of the topology mutation operator based on network distillation [9, 18]. Here we distill the parent's behavior into the offspring, using data $\mathcal{D}^p = (\boldsymbol{s_i}, \boldsymbol{q_i^p})_{i=1}^N$ consisting of N states (or state-action pairs for critic distillation) sampled uniformly at random from the global replay memory, along with the parent network's outputs. This data is then used to perform gradient-based updates on the offspring network using the parent network's outputs as a target in a supervised learning setting:

$$\mathcal{L}(\mathcal{D}^p, \boldsymbol{\theta^o}) = \sum_{i=1}^{|\mathcal{D}^p|} \left\| \boldsymbol{q_i^p} - \boldsymbol{\mu_\theta^o}(\boldsymbol{s_i}) \right\|_2^2 \tag{1}$$

We apply a mean-squared-error loss (MSE) for both distillation updates on the offspring actor and critic, where $\boldsymbol{\mu_\theta^o}$ represents the respective offspring network with parameters $\boldsymbol{\theta^o}$. We use this additional step to stabilize the topology mutation operator, using the parent as a teacher to distill its knowledge into the offspring.

3.2 Gradient-based Mutation

We adopt a second mutation operator (SM-G-SUM) as one of two mutation operators used to evolve the actors in the population. This operator helps creating a more diverse set of actors in the population by altering the parameters of the actor network's layers. Neural network parameter mutations based on Gaussian noise can lead to strong deviations in behavior, often leading to deteriorated performance [19, 20]. In order to stabilize the policies resulting from the mutation operator, we make use of the *safe mutation* operator introduced in [19]. This mutation approach scales the perturbations on

a per-parameter basis, depending on the sensitivity of the network's outputs with respect to the individual parameter.

3.3 Integration in Actor-Critic Neuroevolution

To realize all benefits from the proposed genetic operators as well as the Actor-Critic training, we combine them in the Actor-Critic Neuroevolution (ACN) framework, see Algorithm 1 in the appendix. In the ACN framework, we integrate the two mutation operators described above in a standard GA loop, always mutating the selected candidates by either performing distilled topology mutation (with topology growth probability p_G) or gradient-based mutation (with probability $1-p_G$). After mutating, we add a network training phase for each individual following the setting of Twin-Delayed DDPG (TD3), performing multiple off-policy gradient updates making use of target policy smoothing and clipped double-Q learning [4, 8]. Due to the changes in the neural network architecture the training phase requires a re-initialization of the optimizer and a recreation of the target network at the start of each phase.

The training of each individual can be performed in parallel, since each individual carries its own actor and critic. By adding this training phase, which uses the experiences from a global replay memory, each individual can benefit from a diverse set of policies exploring the environment during fitness evaluation. The training also improves the sample-efficiency of the GA as each offspring receives gradient-based updates, converging to high-reward solutions faster. This is in contrast to purely evolutionary approaches which have to explore the network parameter space in a highly inefficient manner.

4 Experiments

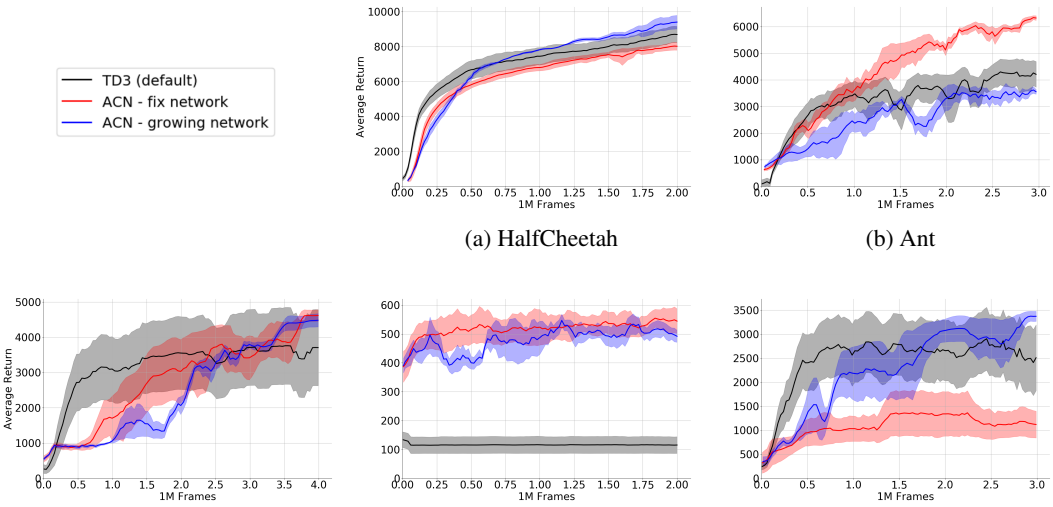
We evaluate ACN on 5 robot locomotion tasks from the Mujoco suite [21]. On these tasks, we compare the performance of a TD3 [8] baseline against two variations of ACN, one which evolves the architecture automatically and one with a fixed network architecture and parameter mutation only. This choice is motivated by the fact that TD3 is the algorithm we employ for each agent's individual training phase in ACN. To the best of our knowledge, this is the first work showing online architecture search on Mujoco tasks. For all evaluated algorithms, we only use a single hyperparameter setting for all Mujoco environments to facilitate comparison with TD3, which was also evaluated with one fixed setting for all Mujoco environments. In terms of network architecture, the fixed architecture algorithms use two layers with 400 and 300 nodes respectively. In case of ACN evolving the topology, we start with a single layer of 64 hidden nodes, initialized using He initialization [22].

Figure 1 shows that, at the end of the optimization, the two ACN variants perform on par with or better than TD3 on all evaluated continuous control tasks. The best architectures found by ACN and the architecture experiments with TD3 are given in Table 1.

Especially in the *Humanoid* environment, the ACN algorithm shows a substantial improvement in performance, which can most likely be attributed to the exploratory nature of the algorithm, both in terms of NN topology and parameters. This is also reflected in the rather atypical architectures found by ACN for this task. In *HalfCheetah*, the final architectures found by ACN are smaller compared to the default architecture. This is consistent with the experiments in Appendix A where a smaller size also outperforms the default architecture. In *Hopper*, ACN takes more environment steps to optimize the network architecture, but eventually catches up, again finding a smaller than usual network size consistent with the findings in Appendix A. The evolved network in *Walker2d* also takes longer to optimize compared with a single TD3 run, but eventually outperforms TD3 with a smaller architecture. The found architecture in *Ant* only contains one layer and half the nodes compared to the TD3 default, but shows comparable performance. In this environment the fixed architecture variant of ACN outperforms TD3. This could be caused by the re-initialization and recreation of optimizer and target networks as shown in Appendix D.

The experiments show the capability of ACN to find suitable network architectures ranging from smaller architectures to larger ones, both in terms of number of layers and the individual layer sizes. ACN achieves this adding only a minor amount of computational cost.

Appendix A shows experiments with different Actor/Critic NN architectures for TD3. These experiments show the significant impact network architecture choices can have on the algorithm's



(a) HalfCheetah (b) Ant

(c) Walker2d (d) Humanoid (e) Hopper

Figure 1: Comparison of mean performance on continuous control benchmarks for ACN with fixed NN topology, with evolving neural network topology and TD3. We used two random seeds for ACN and five random seeds for TD3, the shaded area represents the standard error.

Environment	ACN	TD3 grid search
Hopper	[136, 72]	[200, 150]
Ant	[276]	[600, 450]
HalfCheetah	[80, 80, 88]	[600, 450]
Humanoid	[672, 508]	[400]
Walker2d	[200, 144]	[600, 450]

Table 1: Best actor architectures found by ACN compared with best performing TD3 runs.

performance. We also evaluate the impact of re-initialization of the optimization algorithm and recreation of the target networks during training as it is applied during the ACN network training phase in Appendix D. For that experiment, we apply re-initialization of Adam and recreation of the target networks after each 10k training steps in TD3; it shows that the re-initialization and recreation does *not* tend to have negative impact and sometimes even proves beneficial to the TD3 training.

5 Conclusion

This paper demonstrates how suitable neural network topologies of Actor and Critic networks can be found *online*, while still showing performance comparable with state-of-the-art methods in robot locomotion tasks. We proposed the ACN algorithm, which combines the strengths of Neuroevolution methods with the sample-efficient training of off-policy Actor-Critic methods. To achieve this, we proposed a novel genetic operator which increases the network topology in a stable manner by distilling the parent network's knowledge into the offspring. Additionally, we augmented the GA with an off-policy Actor-Critic training phase, sharing collectively gathered environment interactions in a global replay memory. Our experiments showed that ACN automatically finds suitable neural network architectures for all evaluated tasks which are consistent with strong architectures for these tasks, while only adding a small computational overhead over a single RL run with a fixed architecture.

Further work could investigate the impact of the mutation operator in RL training and why this combination of a GA and RL training often leads to a successful training of smaller topologies while achieving similar or even better performance compared to current RL algorithms.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

[3] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[5] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3357–3364. IEEE, 2017.

[6] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[7] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018. URL http://arxiv.org/abs/1808.00177.

[8] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1582–1591, 2018.

[9] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015. URL http://arxiv.org/abs/1503.02531.

[10] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002. ISSN 1063-6560. doi: 10.1162/106365602320169811. URL http://dx.doi.org/10.1162/106365602320169811.

[11] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009. doi: 10.1162/artl.2009.15.2.15202. URL https://doi.org/10.1162/artl.2009.15.2.15202. PMID: 19199382.

[12] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[13] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[14] Hao-Tien Lewis Chiang, Aleksandra Faust, Marek Fiser, and Anthony Francis. Learning navigation behaviors end to end. *CoRR*, abs/1809.10124, 2018. URL `http://arxiv.org/abs/1809.10124`.

[15] Aleksandra Faust, Anthony Francis, and Dar Mehta. Evolving rewards to automate reinforcement learning. *CoRR*, abs/1905.07628, 2019. URL `http://arxiv.org/abs/1905.07628`.

[16] Frederic Runge, Danny Stoll, Stefan Falkner, and Frank Hutter. Learning to design RNA. In *International Conference on Learning Representations*, 2019.

[17] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[18] Andrei A. Rusu, Sergio Gomez Colmenarejo, Çaglar Gülçehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *CoRR*, abs/1511.06295, 2016.

[19] Joel Lehman, Jay Chen, Jeff Clune, and Kenneth O Stanley. Safe mutations for deep and recurrent neural networks through output gradients. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 117–124. ACM, 2018.

[20] Cristian Bodnar, Ben Day, and Pietro Lio'. Proximal distilled evolutionary reinforcement learning. *CoRR*, abs/1906.09807, 2019. URL `http://arxiv.org/abs/1906.09807`.

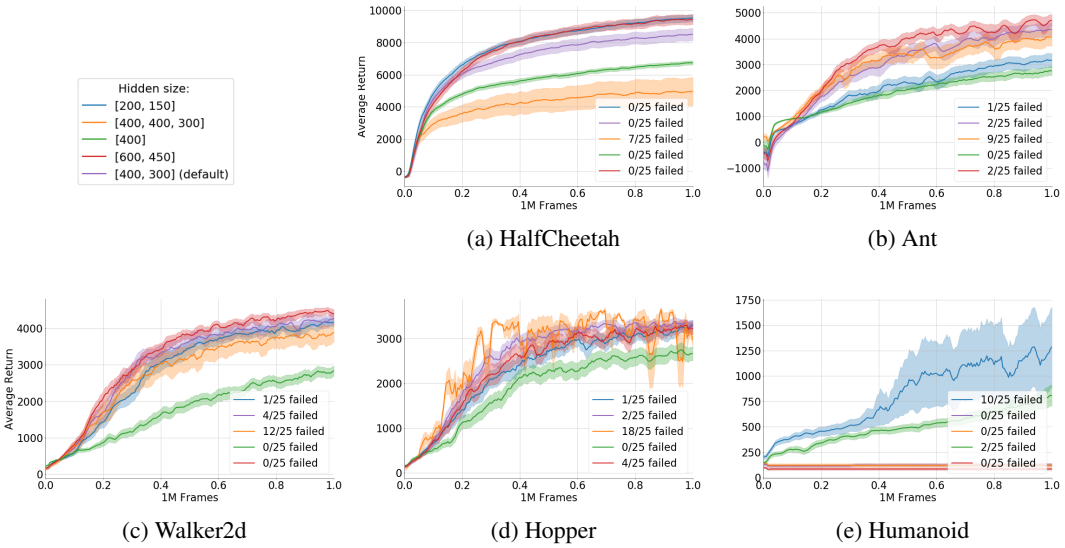
[21] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012. doi: 10.1109/IROS.2012.6386109.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[23] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. *arXiv e-prints*, art. arXiv:1607.06450, Jul 2016.

A Neural Network Architecture Experiments for TD3

We evaluated different choices for the neural network architecture used for both actor and critic networks in the TD3 algorithm in figure 2, keeping all hyperparameters fixed as reported in the original work [8]. Figure 2 shows the results for various neural network architecture choices. Perhaps unsurprisingly, the default architecture chosen in recent literature does not show the best performance on all environments. For example, in the *Humanoid* environment, a simpler topology using only one hidden layer with 400 nodes performs substantially better, while in other environments like *HalfCheetah, Ant* and *Walker2d*, larger capacities like [600, 450] seem to be favorable.



(a) HalfCheetah (b) Ant

(c) Walker2d (d) Hopper (e) Humanoid

Figure 2: Comparison of the mean performance of TD3 with different neural network topologies on MuJoCo continuous control benchmarks. We used 25 random seeds, but we exclude and report the failed runs, where the average return was less than 5% of the best run. The shaded area represents the standard error.

B ACN Algorithm

A high-level description of the proposed ACN algorithm can be seen in Algorithm 1, making use of tournament selection, Actor-Critic training based on TD3 and the combined mutation operator combining gradient-based mutation and the proposed distilled topology mutation (see Algorithm 2).

The EVALUATE function performs a number of Monte-Carlo rollouts in the environment and determines the fitness value as the average cumulative reward across the performed rollouts. The selection operator TOURNAMENTSELECT runs individual tournaments for all actors and critics of the new population, allowing for different Actor-Critic combinations in subsequent children. This choice is made to prevent actors from exploiting their critic's weakness over time, leading to undesired behavior.

After selection, the new population is mutated using transition batches from the global replay memory. With probability p_{growth}, the network architecture is grown. This is achieved either by appending a new layer of the same size as the last hidden layer of the actor (with probability $p_{addlayer}$) or by choosing a number of additional nodes from the set given as a hyperparameter and adding this many nodes to any randomly chosen layer of the architecture. Both architecture growth operators perform identical architecture changes to both the actor and the critic, as indicated by ADDSAMELAYER and ADDSAMENODES in Algorithm 2. The set of possible additional node numbers can be found in Table 2. As this alteration of the network architectures changes the respective network's behavior, we perform network distillation (see section 3.1) updates using transition batches sampled uniformly at random from the global replay memory on both networks. With this additional step, we can distill the respective parent's knowledge into the offspring, thus enabling to grow the network architectures in a

Algorithm 1: Actor-Critic Neuroevolution (ACN) algorithm

Input: population size k, number of generations $\mathcal{G}$

Initialize global replay memory $\Re$
Initialize k individuals $\mathcal{A}_i := \{actor_i, critic_i, fitness_i = None\}$ as initial population $\mathcal{P}_0^*$
for $g = 1$ **to** $\mathcal{G}$ **do**
    $\mathcal{P}_{g-1}, \mathcal{T}_e \leftarrow$ EVALUATE($\mathcal{P}_{g-1}^{\#}$)
    Store transitions $\mathcal{T}_e$ in $\Re$
    $\mathcal{P}_{elite} \leftarrow$ TOPK($\mathcal{P}_{g-1}$)
    $\mathcal{P}_{selection} \leftarrow$ TOURNAMENTSELECT($\mathcal{P}_{g-1}$)
    $\mathcal{T}_s \leftarrow$ sample transitions from $\Re$
    $\mathcal{P}_{mutated}^{\#} \leftarrow$ MUTATE($\mathcal{P}_{selection}, \mathcal{T}_s$)
    $\mathcal{P}_{trained}^{\#} \leftarrow$ ACTORCRITICTRAINING($\mathcal{P}_{mutated}^{\#}, \mathcal{T}_s$)
    $\mathcal{P}_g^{\#} \leftarrow \mathcal{P}_{trained}^{\#} \cup \mathcal{P}_{elite}$
end
// $^{\#}$`Individuals are not evaluated in environment.`

stable manner without requiring additional rollouts in the environment.
Alternatively to growing the network architectures, we mutate the individual actors of each agent in the population with probability $1 - p_{growth}$, making use of the SAFEMUTATION operator described in section 3.2. This mutation operator alters the individual's policy in a stable way, facilitating exploration in the environment.

Algorithm 2: Mutation Algorithm

Input: population $\mathcal{P}$, transitions $\mathcal{T}$

Initialize empty new population $\mathcal{P}_{mutated} = \{\}$
for *each individual* $i \in \mathcal{P}$ *with actor* $\mathcal{A}$ *and critic* $\mathcal{C}$ **do**
    if *random number* < p_{growth} **then**
        if *random number* < *new layer probability* **then**
            $\mathcal{A}_{grown}, \mathcal{C}_{grown} \leftarrow$ ADDLAYER($\mathcal{A}, \mathcal{C}$)
        else
            $\mathcal{A}_{grown}, \mathcal{C}_{grown} \leftarrow$ ADDNODES($\mathcal{A}, \mathcal{C}$)
        end
        $\mathcal{A}_{distilled} \leftarrow$ DISTILLPARENT($\mathcal{A}_{grown}, \mathcal{A}, \mathcal{T}$)
        $\mathcal{C}_{distilled} \leftarrow$ DISTILLPARENT($\mathcal{C}_{grown}, \mathcal{C}, \mathcal{T}$)
    else
        $\mathcal{A}_{mutated} \leftarrow$ SAFEMUTATION($\mathcal{A}$, $\mathcal{T}$)
    end
    $\mathcal{P}_{mutated} \leftarrow$ ADD$\{\mathcal{A}_{mutated}, \mathcal{C}_{distilled}\}$
end
return $\mathcal{P}_{mutated}$

Each offspring created during the mutation phase is then trained individually using the ACTORCRITICTRAINING operator, which follows the off-policy gradient-based updates described in [4], with the extensions introduced in [8]. The trained offspring, along with the elite determined as the best performing individual during evaluation, is then used as the next generation in the GA.

C Hyperparameters

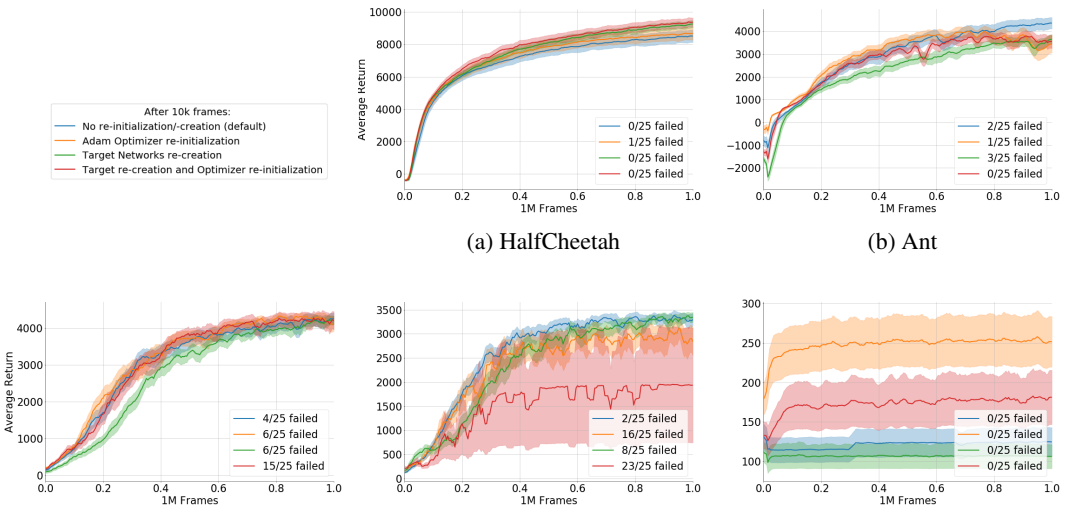
All hyperparameters are kept constant across all environments. For the TD3 training, the same set of hyperparameters as reported in the original paper [8] were used. Table 2 shows the hyperparameters used for ACN across all evaluated environments. All neural networks use the ReLU activation function for hidden layers and linear/tanh activations for critic and actor networks, respectively. We apply Layernorm [23] after each hidden layer as it has proven beneficial for the stability and performance across all experiments in this paper.

Hyperparameter	Value
Population size	20
Elite size	5 %
Tournament size	3
Network growth probability	0.2
Add layer probability	0.2
Add nodes probability	0.8
Set of possible nodes added during layer growth	[4, 8, 16, 32]
Network distillation updates	500
Network distillation batch size	100
Network distillation learning rate	0.1
Safe mutation batch size	1500
Safe Parameter mutation standard deviation	0.1

Table 2: Hyperparameters, constant across all environments.

D Experiments on Re-Initialization of Optimizers in TD3

To assess the performance impact of both the re-initialization of optimizers as inevitably done in ACN, as well as using a new target network after a certain amount of steps, we evaluated different combinations in TD3. Figure 3 shows the impact of different combinations on the performance of TD3 for the continuous control benchmarks used in this paper. Surprisingly, the default TD3 choice does not show the best performance in all environments, as might be expected. Rather, using the current state of the critic as the new target network from time to time seems to benefit performance.



(a) HalfCheetah (b) Ant

(c) Walker2d (d) Hopper (e) Humanoid

Figure 3: Comparison of the mean performance of TD3 with re-initializing the optimizer, re-create the target network or both after 10k frames on MuJoCo continuous control benchmarks. We used 25 random seeds, but we exclude and report the failed runs, where the average return was less than 5% of the best run. The shaded area represents the standard error.