NEUROEVOLUTION OF RECURRENT ARCHITECTURES ON CONTROL TASKS

Maximilien Le Clei & Pierre Bellec
Montreal University Geriatric Institute
{maximilien.le.clei,pierre.bellec}@umontreal.ca

ABSTRACT

Modern artificial intelligence works typically train the parameters of fixed-sized deep neural networks using gradient-based optimization techniques. Simple evolutionary algorithms have recently been shown to also be capable of optimizing deep neural network parameters, at times matching the performance of gradient-based techniques, e.g. in reinforcement learning settings. In addition to optimizing network parameters, many evolutionary computation techniques are also capable of progressively constructing network architectures. However, constructing network architectures from elementary evolution rules has not yet been shown to scale to modern reinforcement learning benchmarks. In this paper we therefore propose a new approach in which the architectures of recurrent neural networks dynamically evolve according to a small set of mutation rules. We implement a massively parallel evolutionary algorithm and run experiments on all 19 OpenAI Gym state-based reinforcement learning control tasks. We find that in most cases, dynamic agents match or exceed the performance of gradient-based agents while utilizing orders of magnitude fewer parameters. We believe our work to open avenues for real-life applications where network compactness and autonomous design are of critical importance. We provide our source code, final model checkpoints and full results at github.com/MaximilienLC/nra/.

1 INTRODUCTION

Artificial neural networks are computing systems that have become central to the field of artificial intelligence. Through waves of innovation, these networks have gotten bigger, more efficient, and increasingly competent on many tasks such as image classification (Dai et al., 2021), image generation (Ramesh et al., 2021) and language modelling (Rae et al., 2021). Most often, the parameters of these artificial neural networks are optimized using first-order gradient-based optimization techniques, yet their architecture is for the most part still constructed manually.

In contrast, many evolutionary algorithm formulations make it possible to not only optimize a neural network's parameters but also construct its architecture (Stanley et al., 2019). And while neuroevolution methods have been overshadowed by gradient-based techniques over the past decade, recent works have shown that evolutionary optimization is competitive with gradient-based learning at optimizing deep neural networks on various reinforcement learning problems (Salimans et al., 2017; Such et al., 2017; Risi & Stanley, 2019). These advances were notably achieved by stripping down many mechanisms popular in traditional evolutionary methods, like agent crossover and speciation, to instead focus on leveraging larger-scale computational resources.

These recent works however only focused on optimizing the neural network parameters, and did not attempt to evolve the network architectures. In this work, we therefore propose to investigate whether evolving neural network architectures can also lead to high performance on popular reinforcement learning tasks. We present a dynamic architecture framework for recurrent neural networks, embedded in a simple evolutionary algorithm, in which agents sample one of four structural mutations every iteration, enabling a population of agents to both increase and decrease their network capacity as the optimization process unfolds.

In order to evaluate these networks of dynamic capacity, we propose to run reinforcement learning experiments on all 19 OpenAI Gym (Brockman et al., 2016) state-based control tasks. We first compare the dynamic networks with standard static-sized deep neural networks optimized through the same evolutionary algorithm and find the dynamic networks generally more efficient and performant than the static networks. And while the static networks that we evolve are already relatively small by modern standards, averaging at ∼8,000 parameters, we find the final elite dynamic networks to be of even lower capacity, averaging at 127 parameters across all tasks. Interestingly, we also find the dynamic networks to be competitive with gradient-based techniques on the majority of these tasks.

2 Related Work

Reinforcement learning (Sutton & Barto, 2018) is an artificial intelligence paradigm where artificial agents learn to maximize some notion of cumulative reward in an environment. Over the past decade, two traditional classes of techniques, Q-learning (Watkins & Dayan, 1992) and policy gradient methods (Sutton et al., 2000), have been most successful in leveraging the representational power of deep neural networks, resulting in some of the most capable reinforcement learning agents to date (Mnih et al., 2013; Lillicrap et al., 2015; Mnih et al., 2016; Schulman et al., 2017; Dabney et al., 2018; Haarnoja et al., 2018; Fujimoto et al., 2018; Wang et al., 2020; Kuznetsov et al., 2020).

In recent years however, various works have shown many settings where, given sufficient computational resources, relatively simple evolutionary algorithms become competitive with gradient-based techniques at optimizing deep reinforcement learning agents. Salimans et al. (2017) first made use of an evolution strategy algorithm in order to optimize deep neural network parameters, resulting in competitive agents on both Atari and MuJoCo tasks. Such et al. (2017) then showed that a simple genetic algorithm, stripped down of popular traditional mechanisms like crossovers and speciation, could also evolve deep neural network parameters in order to play many Atari games and control a humanoid on MuJoCo. Finally, Risi & Stanley (2019) made use of a slightly more complex genetic algorithm in order to optimize the parameters of a World Model (Ha & Schmidhuber, 2018), in doing so evolving a competitive virtual car racing agent.

In this work however, we solely focus on feature-based discrete and continuous control tasks, a setting explored many times over in the field evolutionary robotics (Doncieux et al., 2015). More precisely, we take part in the smaller subset of algorithms evolving both network architecture and parameters. While evolving both of these components has thoroughly been investigated before the turn of the century (Yao, 1999), this field is nowadays best known through the NEAT algorithm (Stanley & Miikkulainen, 2002) for being most capable at solving a multitude of traditional reinforcement learning problems, e.g. the double pole balancing task (Stanley, 2004). Since then, NEAT has been extended to HyperNEAT (Stanley et al., 2009) in order to evolve larger scale neural networks that have been quite successfully applied to various robotics tasks such as evolving the gaits of both digital (Clune et al., 2009) and real-life (Yosinski et al., 2011) quadrupeds, and more recently, extended to evolve a 6-legged digital robot (Huizinga et al., 2016).

Our work is most similar to the NEAT algorithm, but makes the following novel contributions. Our algorithm is first of all less complex, not utilizing speciation and crossover components, and moreover very scalable, conveniently able to leverage large computational resources. The approach is also more general, utilizing fewer hyperparameters, and more flexible, allowing networks to not only increase but also decrease in capacity throughout the optimization process.

3 Dynamic Recurrent Architectures

In this section, we present recurrent neural networks whose architectures dynamically vary in capacity throughout the evolutionary process. Any such network is structured as a directed layered graph and is composed of a variable number of nodes and connections. In the first layer of any dynamic network, input nodes have the potential to transmit input information towards any non-input node located further in the graph. In the last layer, output nodes can potentially receive information from any existing node. Additionally, output nodes get to emit information out of the system and possibly towards any non-input node. Finally, in between these two layers ought to grow hidden nodes that can both receive information from any node and emit towards any non-input node.

In these networks, information flows through forward passes during which, layer after layer, non-input nodes make use of their mutable parameters (weight vector w and bias b) in order to perform linear transformations of their input, followed by non-linear rectifier activations[1]. If a connection between two nodes is headed forward in the graph (meaning towards the output layer), the receiving node gets to make use of the emitting node's latest output information during the same network pass. However, if a connection between nodes is headed backward or to the same layer, the receiving node only gets to use the emitting node's latest output information during the next network pass.

[1] $f(x) = \max(0, w^\top x + b)$

Upon initialization, networks are always set to create the minimal amount of structure to fit the task at hand: a layer of input nodes and a layer of output nodes, both void of connections. In order to model more complex functions, these networks evolve in complexity through four structural mutations:

1) Grow Connection (Section 3.1)
2) Prune Connection (Section 3.2)
3) Grow Node (Section 3.3)
4) Prune Node (Section 3.4)

Before describing these four mutations, we define the following:
In-nodes: *($\neq$ input nodes)* Set of nodes that a given node receives information from. Each node possesses its unique set of in-nodes.
Out-nodes: *($\neq$ output nodes)* Set of nodes that a given node emits information to. Each node possesses its unique set of out-nodes.
Receiving nodes: Set of all input nodes plus all hidden/output nodes that possess in-nodes.
Emitting nodes: List of nodes possessing out-nodes. Nodes appear in this list once per out-node that they possess.

3.1 Mutation #1 : Grow Connection

Step 1. A first node is sampled[2] from the set of all receiving nodes.
Step 2. A second node is sampled from the set of all hidden and output nodes.
Step 3. A new weighted[3] connection is formed from the first to the second sampled node.

We give an example of this mutation in Figure 1.

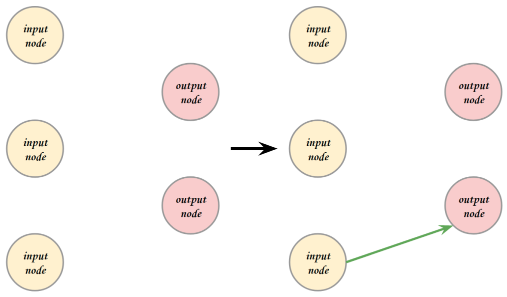

Figure 1: **Mutation #1 : Grow Connection**. In order to showcase the four architectural mutations effectively, we elaborate in the next four figures over a hypothetical scenario in which we require a neural network to input three and output two values. In such a case, a network is initialized with three input nodes and two output nodes, all devoid of connections. Calling the "Grow Connection" mutation on this network requires sampling a first node from the three input nodes and a second node from the two output nodes. Imagining these turn out to be the third input node and the second output node, a weighted connection is created in between the two.

3.2 Mutation #2 : Prune Connection

Step 1. A first node is sampled from the list of emitting nodes.
Step 2. A second node is sampled from the first node's set of out-nodes.
Step 3. The existing connection between the first and second node is deleted.

We give an example of this mutation in Figure 2.

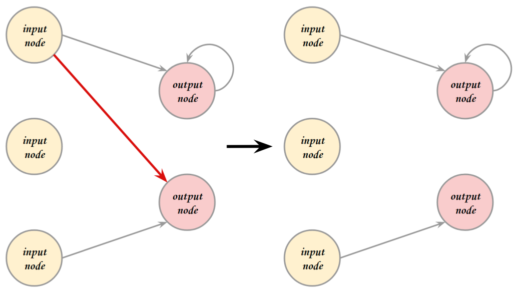

Figure 2: **Mutation #2 : Prune Connection**. We pursue the hypothetical scenario introduced in Figure 1 and assume that three more "Grow Connection" mutations have been applied since. Calling the "Prune Connection" mutation on this network requires sampling a first node from the list composed of the first input node (2x), the third input node and the first output node. In the case that this first node turns out to be the first input node, a second node is now to be sampled from this node's set of out-nodes, set composed of the two output nodes. We now imagine that the second output node is the one sampled. As a result, the connection between the first input node and the second output node is deleted.

[2]In this framework, sampling from sets and lists is always uniform.
[3]Where $\mathcal{N}(\mu, \sigma^2)$ is the Normal distribution with mean μ and variance σ^2, all new network weights and biases are initialized with values sampled from $\mathcal{N}(0, 1)$ and perturbed every iteration with values from $\mathcal{N}(0, .01)$.

3.3 MUTATION #3 : GROW NODE

Step 1. Three nodes are sampled:
- a first node from the set of all receiving nodes.
- a second node from the set of all receiving nodes minus the first node.
- a third node from the set of all hidden and output nodes.

Step 2. A new hidden node is initialized[4].
Step 3. Three new connections are grown:
- from the first node to the new hidden node.
- from the second node to the new hidden node.
- from the new hidden node to the third node.

We give an example of this mutation in Figure 3.

3.4 MUTATION #4 : PRUNE NODE

Step 1. A hidden node is sampled from the set of all hidden nodes.
Step 2. The hidden node and all of its connections are deleted.

Additionally, any hidden node, which as a result of a pruning mutation no longer receives nor emits information is also subsequently deleted.

We give an example of this mutation in Figure 4.

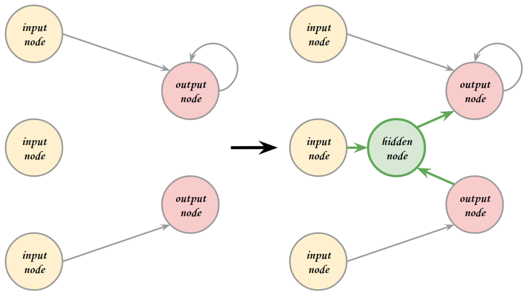

Figure 3: **Mutation #3 : Grow Node**. We continue from where we left off in Figure 2. Calling the "Grow Node" mutation on this network entails sampling 1) a first node from all visible nodes 2) a second node from all visible nodes excluding that first sampled node 3) a third node from the two output nodes. We imagine these turn out to be the second input node, the second output node and the first output node respectively. A hidden node is thus created in between and connections are grown from 1) the second input node to the hidden node 2) the second output node to the hidden node 3) the hidden node to the first output node.

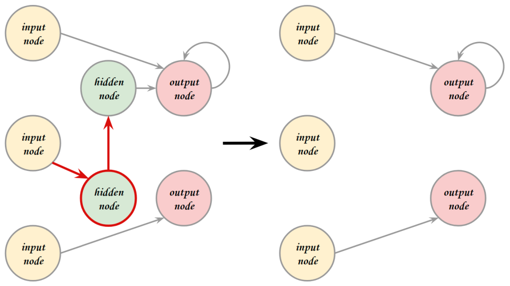

Figure 4: **Mutation #4 : Prune Node**. We now imagine that many of the three mutations described thus far have taken place since Figure 3. Calling the "Prune Node" mutation on this network requires sampling one node from the two hidden nodes. In the case that it turns out to be the second hidden node, it is therefore deleted alongside its two connections. As a result of this deletion, the first hidden node, now devoid of any input information, is also deleted.

Algorithm 1 Neuroevolution Algorithm

```
Input : number of generations $G$, population $P$
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Initialize all agents in $P$
for generation = $1, 2, ..., G$ do
    for agent in $P$ do
        Variation : Perturb the agent's weights and biases
                  + Sample and apply 1 architectural mutation (dynamic network populations only)
        Evaluation : Run the agent and track its fitness
    end for
    Selection : $P \leftarrow$ Agents with top 50% fitness duplicated
end for
```

[4]In our implementation, the new hidden node is always positioned one layer past the first in-node, towards the out-node. If no layer exists between those two nodes, a new one is created to fit the new hidden node.

4 Experiments

In order to evaluate our methods, we propose to run experiments on all 19 state-based control task environments currently available through the OpenAI Gym library (Brockman et al., 2016). Throughout these environments, agents get to control virtual bodies like simplified robotic arms, vehicles and various types of androids. In order to perform the control tasks, agents are iteratively fed input values characterizing various pieces of information such as the position, angle and speed of their body parts. In turn, agents are expected to output values representing various actuation forces onto their body components. As a means to drive behaviour in reinforcement learning settings, the environments define termination criteria and reward signals which we both make use of in order to evaluate our agents.

To optimize the agents, we set up a simple neuroevolution algorithm as described in Algorithm 1. In order to better understand the value brought forward by our framework, we propose to not only evolve dynamic networks but also standard deep recurrent neural networks of dimensions $[d\_input, 50, 50, d\_output]$, similar to the networks described by Salimans et al. (2017) for their MuJoCo experiments.

After choosing a population size and a task to optimize for, the evolutionary optimization process begins by initializing a population of network agents. While no architectural operation is required in static-sized deep neural networks, the dynamic networks are always set to first create the minimal amount of structure to fit the task at hand, as described in Figure 1. Then, regardless of dynamicity, all network weights and biases are initialized to zero. The evolutionary process then starts iterating over three distinct stages commonly referred to as variation, evaluation and selection.

First, during the variation stage, we begin by randomly perturbing the parameters of both dynamic and static networks. Like for dynamic networks, we propose to perturb static networks with values drawn from the normal distribution $\mathcal{N}(0, 0.01)$. As described in Section 3, dynamic agents then sample one of four architectural mutations, which they in turn apply to their existing network.

Then, during the evaluation stage, agents run one episode[5] in the environment until termination. In environments requiring continuous action values, we clip the ReLU activated values emitted from the networks' output nodes to the range [0, 1] and scale them to the expected range of outputs. In environments requiring discrete action values, we instead retrieve the position of the output node emitting the largest value. Lastly, while we directly feed agents with values emitted from the environment in most tasks, early experiments found the distribution of input values to hinder progress in certain environments. We therefore put in place a running standardization of inputs for classical task *Pendulum* and MuJoCo tasks *Ant*, *HalfCheetah*, *Hopper*, *Humanoid*, *Reacher* and *Walker2d*.

Finally, during the selection stage, we utilize a 50% truncation strategy, in which agents with upper half performance in terms of accumulated reward are maintained and duplicated. Agents then proceed back to the variation stage and keep iterating until termination. We fully parallelize the variation and evaluation stages of our algorithm and run the entirety of these experiments on a cluster with nodes powered by AMD 7532 CPUs, connected through NVIDIA QM8700 switches. We provide information about computation speed in Table 1 plus the source code, final model checkpoints and results at github.com/MaximilienLC/nra/.

5 Results

We display in Figures 5 to 8 the evolution of scores achieved by various static and dynamic network populations on all 19 tasks. These scores were obtained by averaging performance over 10 newly seeded runs[6]. We complement these figures with a representation of the final dynamic elite (highest scoring) architecture for each task.

We first observe that across all tasks, neural network agents with dynamic architectures always match and often even outperform (ex: *BipedalWalker* and *Reacher*) evolved static deep neural network agents. Moreoever, we find cases in which dynamic agents require smaller population sizes (ex: *MountainCar*), a lower amount of iterations (ex: *Acrobot* and *CartPole*) or even both (ex:

[5]Except for environments *Pendulum* and *Reacher*, in which we run five episodes to more accurately evaluate overall agent performance given their large task initialization variance.

[6]We iterate over seeds starting from 0 during the evolutionary optimization and make use of seeds 2^{31}-10 to 2^{31}-1 at test time.

InvertedDoublePendulum and *InvertedPendulum*) in order to reach what we believe to be optimal performance. We also find cases where static network agents, as opposed to dynamic agents, no longer appear to be capable of making sustained progress (ex: *Ant* and *HalfCheetah*).

Secondly, and perhaps more importantly, we find that out of the 19 control tasks, the dynamic network agents either match or surpass the majority of the baselines on 13 tasks. Out of the remaining 6 tasks, dynamic agents still appear to be trending upwards (at varying speed) after 5000 generations on 5 tasks but seem unable to make further progress on the task *BipedalWalkerHardcore*.

Finally, we find from the third column of these figures that the number of parameters utilized by elite dynamic agents is extremely condensed by modern standards, ranging from 3 parameters on *MountainCarContinuous* to 322 parameters on *Pendulum*, averaging at about 127 parameters across all tasks. We notice quite different architectures evolved across different tasks, some not requiring hidden nodes (ex: *CartPole* and *MountainCarContinuous*), some that are much deeper than they are wide (ex: *HalfCheetah* and *Reacher*) and others with early layers much wider than later ones (ex: *Ant* and *Humanoid*). Generally though, network layers tend to be very interconnected through both forward and recurrent connections.

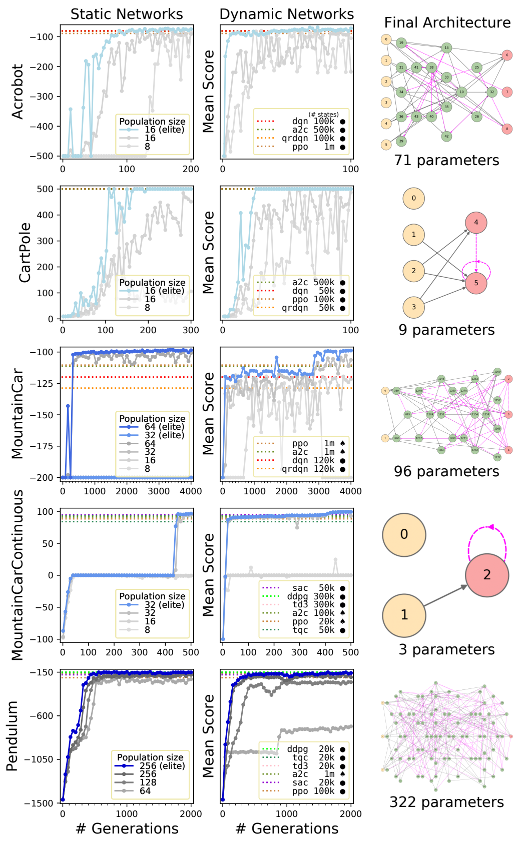

Figure 5: **Classic Control Tasks Results**. (Baseline results marked with ● are reported from evaluating pre-trained agents Raffin (2018) on the same seeds as our agents. Baseline results marked with ♠ are directly reported from Raffin (2018)). Both static and dynamic networks match or surpass the baselines on classic control tasks.

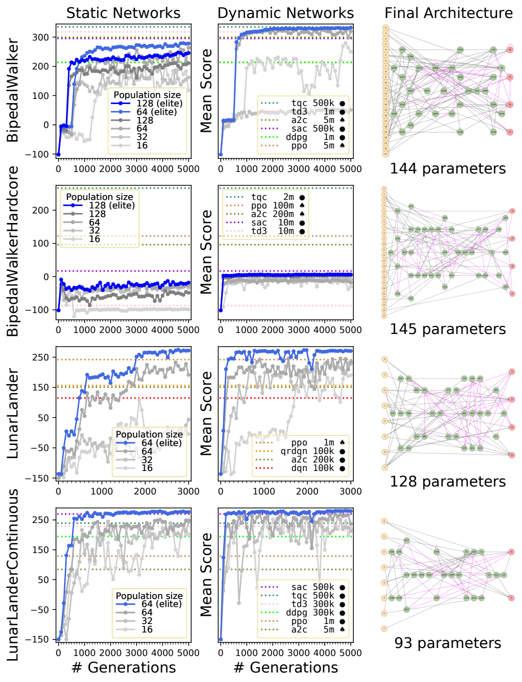

Figure 6: **Box2D Control Tasks Results**. Dynamic networks perform as well or better than static networks on all Box2D tasks. They are also competitive with the baselines on all tasks except *BipedalWalkerHardcore*.

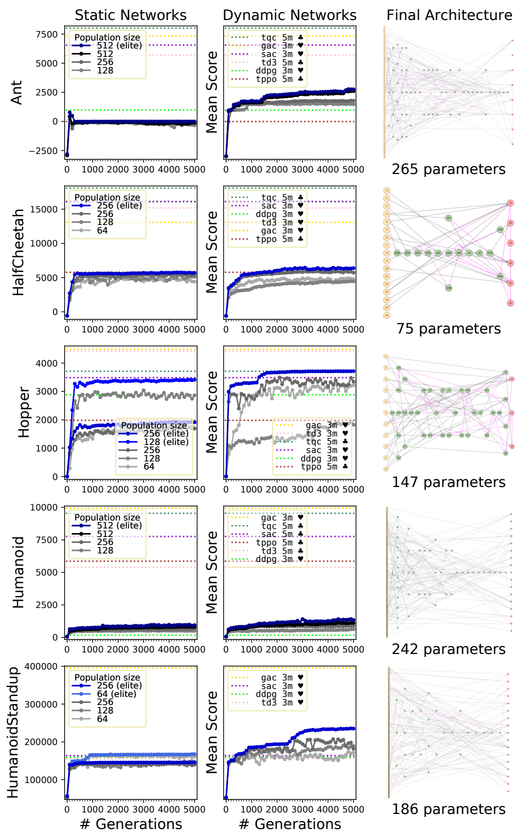

Figure 7: **MuJoCo Control Tasks Results (1/2)**. (Baselines marked with ♣ and ♥ are reported from Kuznetsov et al. (2020) and Lingwei (2021) respectively) Dynamic networks perform better than static networks across this first batch of MuJoCo tasks. Performance matches or exceeds most of the baselines on *Hopper* and *HumanoidStandup*. Progress is slower on *Ant*, *Half-Cheetah* and *Humanoid*, not quite reaching the same level of performance in 5000 generations.

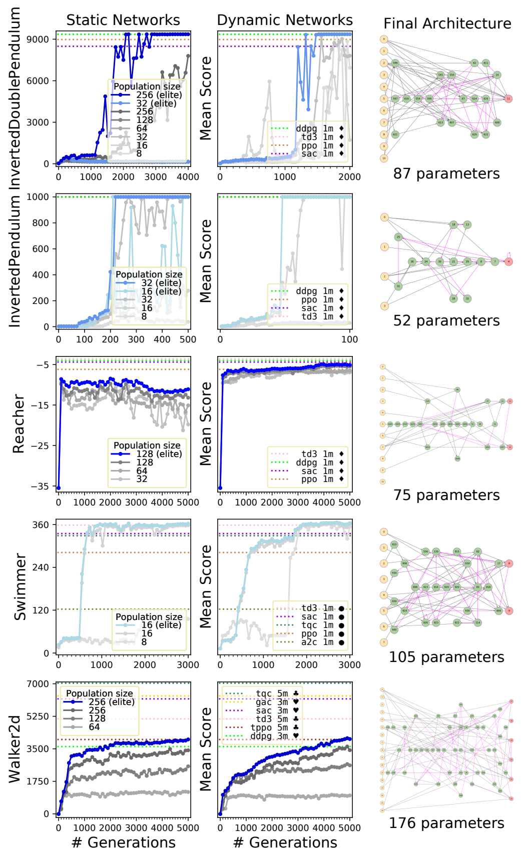

Figure 8: **MuJoCo Control Tasks Results (2/2)**. (Baselines marked with ♦ are reported from Fujimoto et al. (2018)) Dynamic agents perform as well or better than static agents on the remaining MuJoCo tasks. Performance comes close to matching, matches or exceeds the baselines on all tasks except *Walker2d* where it has yet to surpass the majority of theses techniques after 5000 generations.

6 Discussion

We have presented in this work recurrent neural networks whose architectures dynamically evolve in capacity during the optimization process. We made use of a simple yet scalable evolutionary algorithm and experimented on many continuous and discrete control tasks. We first found this framework to evolve networks that are generally more capable than evolved static-sized networks, leading us to believe that starting off with the smallest possible architecture and progressively constructing it enables agents to better adapt to each of these tasks. In addition, we found the dynamic networks to have very interconnected layers and various types of recurrences which appear to grant networks more representational flexibility and thus have them require utilizing fewer parameters. Since we did not perform any hyperparameter search, it is quite possible that static networks would perform better with different hyperparameters on many of these tasks. However it seems equally likely that dynamic networks would also benefit from task specific hyperparameters such as sampling more mutations per iteration in complex MuJoCo tasks. The focus of this work was rather to explore whether a non-specialized approach could perform well across many tasks.

We then found the dynamic architecture framework (and the static framework to a lesser degree) to produce network agents that are most often competitive with the deep reinforcement learning agent baselines. And while progression remains quite slow for several tasks, not quite reaching most baselines' performance after 5000 generations, their progression asymptote often suggests that the evolutionary optimization has yet to converge. In addition to longer run times, results from Such et al. (2017) that surpass the majority of the baselines presented in this work on the task *Humanoid* seem to suggest that larger population sizes and more specialized hyperparameters could provide further improvements. Finally, we remark that modern reinforcement learning techniques are often evaluated in terms of data efficiency and therefore cannot guarantee that these baselines correspond to some of these agents' optimal performance. Our technique often requires running a much larger number of environment steps and it could be argued that deep reinforcement learning algorithms would make better use of running these many states. However, evolutionary algorithms such as the one used in this work are by design very straightforward to massively parallelize and can therefore attain this amount of states in reasonable run times without even requiring the use of GPUs.

Importantly however, as Such et al. (2017) discovered in their Atari experiments, we encounter tasks (e.g. *BipedalWalkerHardcore*) in which agents are noticeably unable to make sustained progress over thousands of generations, which further suggests that as a drawback of its simplicity, this evolutionary optimization formulation is considerably affected by unfavorable reward landscapes that are too coarse to drive the desired change in behaviour. Exploring other evolutionary learning paradigms to produce behaviour in such settings therefore seems like a worthwhile future direction.

7 Conclusion

In addition to joining earlier works in suggesting that evolutionary algorithms are a competitive alternative for optimizing neural network parameters, we believe our work to be a new step forward in demonstrating the feasibility of evolving network architectures in complex machine learning settings. We find clear advantages to constructing architectures like requiring orders of magnitude fewer parameters in order to reach very high performance on many reinforcement learning state-based control tasks. And while our results do not achieve state-of-the-art performance on every task that we experiment on, we believe the conceptual simplicity of our method to give room for many types of improvements. Moving forward, leveraging the continually expanding representation space of such dynamic networks could prove to be valuable in order to explore both continual and multi-task learning settings. In the meantime, we believe that our framework could prove useful in real-life applications where network compactness and autonomous design are of critical importance.

Table 1: **Runtime, number of visited states and elite behaviour of the largest dynamic network populations on each task.**

Task	Runtime (cores)	States	Behaviour
Acrobot-v0	0h01m (16)	~138K	[video]
CartPole-v1	0h02m (16)	~657K	[video]
MountainCar-v0	0h06m (32)	~15M	[video]
MountainCarContinuous-v0	0h02m (32)	~4M	[video]
Pendulum-v1	2h13m (256)	~512M	[video]
BipedalWalker-v3	3h01m (64)	~419M	[video]
BipedalWalkerHardcore-v3	3h28m (128)	~1B	[video]
LunarLander-v2	1h15m (64)	~48M	[video]
LunarLanderContinuous-v2	3h16m (64)	~62M	[video]
Ant-v3	2h49m (512)	~3B	[video]
HalfCheetah-v3	1h56m (256)	~1B	[video]
Hopper-v3	2h35m (256)	~1B	[video]
Humanoid-v3	2h00m (512)	~532M	[video]
HumanoidStandup-v2	7h04m (256)	~1B	[video]
InvertedDoublePendulum-v2	0h09m (32)	~21M	[video]
InvertedPendulum-v2	0h01m (16)	~872K	[video]
Reacher-v2	0h31m (128)	~160M	[video]
Swimmer-v3	0h49m (16)	~48M	[video]
Walker2d-v3	3h00m (256)	~1B	[video]

REFERENCES

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Jeff Clune, Benjamin E Beckmann, Charles Ofria, and Robert T Pennock. Evolving coordinated quadruped gaits with the hyperneat generative encoding. In *2009 iEEE congress on evolutionary computation*, pp. 2764–2771. IEEE, 2009.

Will Dabney, Mark Rowland, Marc G Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *arXiv preprint arXiv:2106.04803*, 2021.

Stephane Doncieux, Nicolas Bredeche, Jean-Baptiste Mouret, and Agoston E Gusz Eiben. Evolutionary robotics: what, why, and where to. *Frontiers in Robotics and AI*, 2:4, 2015.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pp. 1587–1596. PMLR, 2018.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

Joost Huizinga, Jean-Baptiste Mouret, and Jeff Clune. Does aligning phenotypic and genotypic modularity improve the evolution of neural networks? In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 125–132, 2016.

Arsenii Kuznetsov, Pavel Shvechikov, Alexander Grishin, and Dmitry Vetrov. Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In *International Conference on Machine Learning*, pp. 5556–5566. PMLR, 2020.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Peng Lingwei. Generative actor-critic: An off-policy algorithm using the push-forward model. *arXiv preprint arXiv:2105.03733*, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

Antonin Raffin. Rl baselines zoo. `https://github.com/araffin/rl-baselines-zoo`, 2018.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

Sebastian Risi and Kenneth O Stanley. Deep neuroevolution of recurrent and discrete world models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 456–462, 2019.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

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.

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

Kenneth Owen Stanley. *Efficient evolution of neural networks through complexification*. The University of Texas at Austin, 2004.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000.

Yuhui Wang, Hao He, and Xiaoyang Tan. Truly proximal policy optimization. In *Uncertainty in Artificial Intelligence*, pp. 113–122. PMLR, 2020.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

Jason Yosinski, Jeff Clune, Diana Hidalgo, Sarah Nguyen, Juan Cristobal Zagal, Hod Lipson, et al. Evolving robot gaits in hardware: the hyperneat generative encoding vs. parameter optimization. In *ECAL*, pp. 890–897, 2011.