Distributed Evolution

Mario Srouji and Josh Payne

June 2020

1 Introduction and Related Work

Reinforcement Learning (RL) allows unsupervised learning of policies to accomplish tasks in widely different application areas, and it has been proven practically effective when coupled with non linear function approximators, like DNNs [11, 10]. Together with the creation of new RL algorithms and the successful re-implementation of old approaches, methods for accelerating RL have been under investigation in the last years. On the one hand, researchers proposed RL algorithms with improved sample efficiency (e.g. [1, 15, 12, 18]), but also generally more complicated and less prone to be analyzed from a theoretical point of view [8]. On the other hand, significant acceleration has been achieved by efficient schemes that avoid time costly data movements [2, 3] and through distributed implementations of RL algorithms [5, 16].

Simple Evolutionary Strategies (ES) have been recently proven to be competitive, although not yet at par, with RL [4, 7, 8, 14, 17]. When compared to RL, ES can be interpreted as derivative-free direct-policy optimization methods, which either do not make use of the gradient at all (like in the case of genetic algorithms [17]) or approximate the gradient with finite differences [14]. ES have several potential advantages: since the fitness evaluation is independent from action frequency and delayed rewards, ES are also tolerant to extremely long horizons; furthermore, their computational cost is not affected by the computation backward passes, temporal discounting, and value function evaluation [14]. Lastly, ES also scale naturally to distributed systems. This fact represents a fundamental aspect for the development of effective ES strategies, if one also considers that the effectiveness of ES is strictly related to the population size and the high chance of finding effective solution even with random search [8] — in other words, ES are effective only if the population size is large enough, and the limited memory footprint and computational cost of ES guarantees exactly this: that a large population can be used. The drawback of ES is that, since fitness evaluation, and consequently policy update, requires an entire episode to be completed by the entire population, ES are generally less sample efficient than RL, where policy update can occur at high frequency while training, e.g. by bootstrapping from the current estimate of the value function in model-free temporal difference RL [10]. Overall, despite the smaller computational footprint partially offset the worse computational efficiency, ES are still not effective as RL in terms of wall-clock time [14, 8].

We show that the computational efficiency of ES can be significantly improved through a simple yet effective distributed strategy.

2 Distributed Evolutionary Strategies

Generally speaking, gradient-based approaches are difficult to parallelize between machines in a fault-tolerant manner [9]. However, this drawback does not extend to ES. Unlike gradient-based methods, which need a somewhat unified and siloed architecture to practically perform gradient descent, search strategies like ES are designed in such a way that parallelization is a natural implementation. The basic premise is that population members which perform well are selected to be modified and perturbed, whereas lower-performing population members are dropped. Where population members are trained and how they interact with the ground truth gradient with respect to the other population members does not matter; a policy that selects population members that perform well is at the core of ES and can easily be executed by a distribution of machines.

2.1 Distributed Training

The core idea behind distributed evolutionary strategies is that a central leader machine will select the best performing models for further perturbation and training by worker machines. The computationally intense aspect of ES is the procedure done on the local worker machine, and because exploration is randomized, work is not duplicated. The central leader machine sends out the best performing models to the worker machines, which perform a set amount of iterations of training on this model, marginally improving the most performance and moving it in some direction on the gradient before returning the resulting model to the leader. The leader will then compare each of the models it has received from all of the workers, selecting the best performing one with respect to the fitness score as ascertained from the model's performance on a predefined environment, and proceed with the training iterations with that model. This iteration process continues until either the maximum number of iterations has been reached, or until the model has a satisfactory performance; whatever happens first. Distributed methods for ES have been recently proposed [**?**] which rely on passing models between machines. As you might imagine, however, passing weights between members of a network can put a larger-than-is-desirable load on the network, resulting in high latency. To minimize data passing from one machine to another, we observed that the seed used to generate the noise matrix used to perturb the model is sufficient to convey the current trained state of a model, given its previous weights. That is, if one has a fully trained model, they can reconstruct this model using the original set of untrained weights and the sequence of seeds used to generate noise matrices to perturb these weights.

Passing seeds instead of entire weight sets dramatically reduces the amount of data that is required to be passed across the network. The complete algorithm for seed-passing-based distributed evolutionary strategies is shown in Algorithm 1.

2.2 Leader Election

We now have a method that trains a model over a distributed network of machines using a paradigm of a leader and workers. Once a leader has been determined, workers can come and go as they will without an issue – the leader simply selects the top-performing models of all the models it receives back from the workers for a given iteration. In this sense, the system is very worker-fault-tolerant. However, what happens when the leader fails? In this case, we leverage the leader election procedure as outlined in RAFT [13]. In this case, a machine is designated as a leader, and the rest as followers. The leader is responsible for

Algorithm 1: Distributed ES

Input: Leader, Workers, number of iterations i, environment e, noise matrix generator $gen()$, population size p

Output: Trained model M

Note: assume model is a function, its output is a fitness function, and we want to maximize the fitness function.

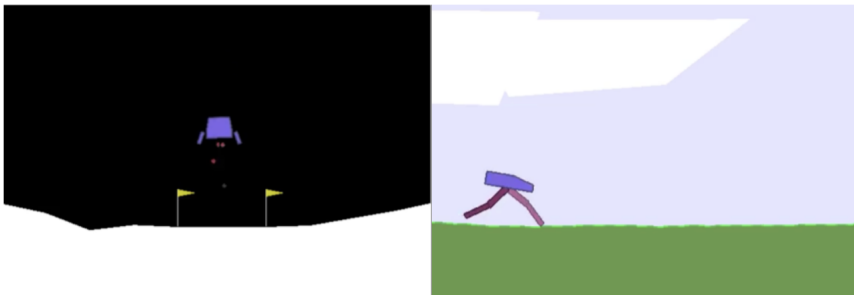
Each of Workers generates duplicated M from random
Leader generates seed $curr\_seed$
for i iterations:
    Leader creates empty list $fitnessTuples$
    for each worker in Workers:
        Leader sends $curr\_seed$ to worker
        worker updates $M \leftarrow M * gen(curr\_seed)$
        worker generates try_seeds, list length p of random numbers
        worker creates model fitness tuple $workerFitnessTuples$
        for j in range(p):
            $M_j \leftarrow M * gen(try\_seeds[j])$
            $f \leftarrow M_j(e)$
            worker appends $(try\_seeds[j], f)$ to $workerFitnessTuples$
        worker sends max($workerFitnessTuples$) w.r.t. f back to Leader
    Leader appends (try_seed, f) from each worker to $fitnessTuples$
    $curr\_seed$ = max($fitnessTuples$) w.r.t. f
    Leader updates $M \leftarrow M * gen(curr\_seed)$
Leader returns M

model comparison, selection, and distribution as before, and the followers are responsible for training, as before. The leader will, parallel with training, send out a heartbeat to all of its followers. If the heartbeat times out on one of the followers, it changes its status to "candidate". Upon joining, a new worker receives the current seed sequence from the leader. Further, all workers are trusted to have the current seed sequence. If any of the workers times out, it becomes a candidate and issues a RequestVotes RPC with a term number. Randomized election timeouts are used to prevent split votes. If a worker is elected, it broadcasts this to the other workers and becomes the leader. Training then continues with the elected candidate as the leader.

A simpler leader election method is also proposed for a trusted distributed system: timeouts are determined by end-of-iteration cycle time. This is different for each node, as training does not have a set runtime. The machine which fails to connect to the predefined leader port after completing its iteration first assumes the role as leader and immediately opens up this port to other workers, which return their seeds to this new leader on the network.

Loss of a leader does not mean loss of a model, as each iteration ensures that the entire system is updated with the latest weight set to iterate on. The big idea here is that such a system could be "released" into the world, with any machine joining the network given the latest weights and able to contribute easily to the training. As long as at least one machine remains on the network, the training progress is not lost.

3 Experimental Results



Compared with previous work, the synergy within the system enables state of the art results on multiple locomotion simulations from OpenAI gym and MuJoCo. To demonstrate the general applicability and effectiveness of our approach, we experiment across a diverse set of environments, including locomotive and control tasks with both discrete and continuous actions spaces. We avoid any environment specific hyper-parameter tuning, and fixed all hyper-parameters throughout the training session and across experiments. In addition, favoring random seeds has been shown to introduce large bias on the model performance [6]. Hence for fair comparison we also avoid any specific random seed selection, running each experimental setup across 5 different random seeds.

	DisES	**ES**	**DDPG**
LunarLander	0.67	5.67	45
BipedalWalker	9.5	47	240
Ant	70	778	406
HalfCheetah	326	672	762
Hopper	68	302	278

The table above shows the training time in **minutes** until convergence (learning the task to the defined reward-threshold), for each environment and training algorithm. Our Distributed Evolution algorithm is listed in the first column, vanilla ES in the second column, and Deep Deterministic Policy Gradients (DDPG) in the third column. As we show in the results, our distributed implementation not only is faster than the standard evolutionary strategies baseline, but it is also much faster than a competitive policy-gradient based algorithm in RL. It is notable that vanilla ES has on average larger training times than policy gradient algorithms in RL, however with our approach we are able to significantly reduce the wall-clock time required to learn these complex tasks (at times by an order of magnitude). This is due to the enhanced exploration of the algorithm, combined with the ability to spawn much larger population sizes due to the cooperative nature of the clients in the implementation.

4 Conclusion

In this work, we proposed a novel method of performing evolutionary strategies (ES) algorithms on a distributed system using seed passing and leader election for leader fault tolerance. We discussed why this procedure gives benefits on top of single-machine ES as

well as gradient-based methods and have supported this discussion with encouraging experimental results. For next steps, we plan to fine-tune our methods, using linear combinations of models instead of max selection for model selection, as well as explore group ES in the distributed setting for even higher performance. Ultimately, if a distributed ES system could learn a complex task such as the game of Go, it would be a monumental breakthrough and indicate its viability as a scalable alternative to gradient-based methods in deep learning.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5048–5058. Curran Associates, Inc., 2017.

[2] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. In *ICLR*, 2017.

[3] Steven Dalton, Iuri Frosio, and Michael Garland. Gpu-accelerated atari emulation for reinforcement learning. *CoRR*, abs/1907.08467, 2019.

[4] Madalina M. Drugan. Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms. *Swarm and Evolutionary Computation*, 44:228 – 246, 2019.

[5] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.

[6] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Shauharda Khadka, Somdeb Majumdar, Tarek Nassar, Zach Dwiel, Evren Tumer, Santiago Miret, Yinyin Liu, and Kagan Tumer. Collaborative evolutionary reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3341–3350. PMLR, 2019.

[8] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *NeurIPS*, 2018.

[9] Konstantin Mishchenko, Filip Hanzely, and Peter Richtárik. 99% of parallel optimization is inevitably a waste of time. *CoRR*, abs/1901.09437, 2019.

[10] 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 Maria Florina Balcan and Kilian Q. Weinberger,

editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[12] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6292–6299, 2018.

[13] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[14] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[16] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning, 2018.

[17] 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*, abs/1712.06567, 2017.

[18] Denis Yarats, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, and Rob Fergus. Improving sample efficiency in model-free reinforcement learning from images, 2019.