Artificial Intelligence Report

Lucille Blumire

Contents

INTRODUCTION

Evolution strategies have been a known optimisation technique since the 1960s and 70s. They are based on principles of evolution through continuous improvement via artificial selection to maximise a given function. Despite their relative age, in 2017 a paper by Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever was published under the OpenAI organisation [1]. This paper explored the idea that the relatively ancient technique of using evolution strategies for optimisation might give comparable results to more modern reinforcement learning techniques in a number of problem domains. In this report I'll be analysing the paper and findings in it, as well as the associated blog post [2].

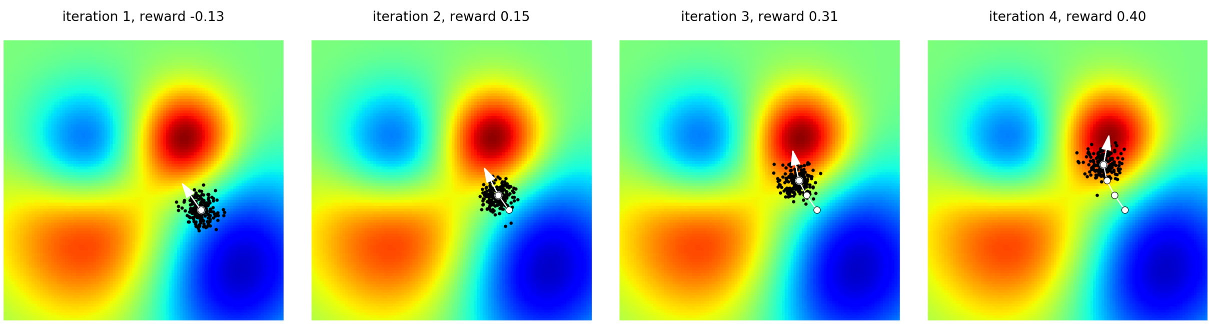
The approach taken by OpenAI is to use evolution strategies to optimise a neural network which takes in an environmental state and outputs the desired action or actions for an agent in that environment to take.

In a more modern reinforcement learning situation this might be done through backpropagation, where various iterations are run of simulating the environment, and each time the agent takes an action the weights of the neural network are adjusted through backpropagation to make that action more or less likely depending on the success of the network.

The approach taken by OpenAI here is different however, in that it instead uses evolution strategies to define a policy function in which a vector representing the entire state of the underlying neural network are used as inputs, and a single reward number is given as an output. These inputs are then modified and altered through evolution strategies to attempt to maximise that reward output, rather than back-propagation the reward to adjust weights in the network directly.

This fashion of updating a neural network has several obvious immediate benefits, and that is that given enough randomness to the selection and evolution process, and enough possibility for mutation, the network will be able to find a global maximum (given enough time) rather than simply some good-enough local maximum to the reward function. In the case of the latter, some reinforcement techniques such as backpropagation can find a local maximum to their reward, in which any significant change to the network would result in a decreased reward and so is undesirable. The former means that through evolution strategies, it is possible to escape a local

Fig. 1 ES optimisation process, with only two parameters and a reward function (red = high, blue = low). At each iteration the current parameter value is white, the Gaussian samples are in black, and the estimated gradient is the white arrow.



maximum through substantial mutation. Evolution strategies will still however often find themselves selecting a local maximum, but the chance of this can be decreased by increasing the extremeness of mutation.

A visual intuition for how the evolutionary strategies discussed work can be seen in Figure 1 (page 3) which depicts the gradual convergence on a local maximum in a given field of pottential values.

Over the course of this report, vectors are used heavily in notation, as well as vectors of vectors. As such, notationally I will be using the following.

$\vec{A}$	is a vector.
$\vec{B}_j$	is the scalar at index j in the vector B where B is a vector of scalars.
$\vec{C}^i$	is the vector at index i in the vector C where C is a vector of vectors.
$\vec{C}^i_j$	is the scalar at index j in the vector $\vec{C}^i$ where C is a vector of vectors of scalars.

1 The Problem Addressed

A number of issues are addressed by using Evolution Strategies over more modern Reinforcement Learning strategies. The first and foremost of these is completely eliminating the need for backpropagation or value function estimation. This aids in writing code as it can be kept simpler and much easier to understand, and allows for exploring non differential networks.

The next problem that is addressed and possibly the most significant, is that it is notoriously difficult to parallelise backpropagation [3]. This is not the cast with Evolution Strategies, which are extremely easy to parallelise. As is explained in the paper and summarised in the blog post, Evolution Strategies depend on only a few small scalar values to be communicated between parallel execution agents. This massively reduces the amount of synchronisation that is required, and enables Evolution Strategies to be run on a far larger distributed computation cluster without diminishing returns when compared with backpropagation reliant reinforcement learning.

Another problem with reinforcement learning strategies is that often they will take a long time to begin producing at all functional outputs, let alone optimal ones. This is due to its stochastic solution space exploration, and is somewhat mitigated by techniques used in Q-Learning where agents can be made to perform consistent action rather than indecision or contradictory action. This issue is mitigated entirely by evolutionary strategies as they can use a structured exploration with deterministic policies enabling for immediate results obtained in a consistent fashion, with an eventual trend towards a locally optimum network.

Value estimation can often be difficult in some problem spaces, and this can result in an unclear means of determining success and providing sensible feedback to the neural network with reinforcement learning. This also occurs when actions have long lasting effects and the time window for an episode is long in both opportunity for action and cumulative in consequence of action. This is a common scenario in many video-game playing applications, or robotic agents interacting with their environment. Evolutionary Strategies are able to provide a better gradient to an optimum solution in these situations.

2 Mathematical Model of Learning and Gradiant Ascent

The objective function for evolution strategies is a function that takes knowledge of its computation environment, and a set of configurations for a neural network. It internally constructs the neural network and then produces a correctness or reward value for the given output of its internal neural network. This might be a direct match against known categories for training a categorisation problem, or might be the outcome of playing a simple game or even a more complex game (such as the NES games used to test the effectiveness of the system in the paper as demonstrated by the blog post).

By having the objective function being optimised use the weighting of a neural network as inputs, the neural network is optimised by review of its weights and output, not by its input and output. This eliminates the need for backpropogation and provides a simple and easily understood objective function and optimisation learning process.

The class of evolution strategies used to optimise the objective function are called natural evolution strategies [4] [5] [6] [7] [8] [9].

Algorithm 1 (page 7) is derived from approximating samples from the following score function estimator to optimise over $\vec{\theta}$.

$$\nabla_{\vec{\theta}} \mathbb{E}_{\vec{\epsilon} \sim \mathcal{N}(\vec{0},I)} \{F(\vec{\theta} + \sigma\vec{\epsilon})\} = \frac{1}{\sigma} \mathbb{E}_{\vec{\epsilon} \sim \mathcal{N}(\vec{0},I)} \{(F(\vec{\theta} + \sigma\vec{\epsilon}))\vec{\epsilon}\} \quad 1$$

Understanding the above mathematical formula requires understanding a high degree of syntax, so I will state it in words as follows, which might then help with understanding.

In the above, F represents the objective function. This takes in a vector of weights a parameter, uses them to construct a neural network, tests that neural network against a given simulation, and returns a scalar reward value representing the effectiveness of that network.

Theta is then the vector of weights that are passed in to construct the neural network.

Epsilon is used in all contexts as representing a vector of random Gaussian noise that can be applied to the weights of the network. It has this context because it is always used following $\mathbb{E}_{\vec{\epsilon} \sim \mathcal{N}(\vec{0},1)}$ explained further down.

Sigma represents a known scalar standard deviation which is set and can be adjusted to make the effective area of the search space of the Gaussian blurred weights larger or smaller. This can be visually understood as understanding that in Figure 1 (page 3) a larger standard deviation would result in the area encompassed by the cluster of black dots being larger, and a smaller standard deviation would result in a more focused cluster of black dots.

$\mathbb{E}$ is used when analysing a stochastic function such as the evaluation of a neural network with given weights, to take the expected value of the function. This in a way can be thought of as a weighted average of its possible values.

$\nabla_{\vec{\theta}}$ simply expresses that the final value we are computing is the derivative or slope of the function with respect to $\vec{\theta}$, which is then used with gradient ascent to perform optimisation and learning.

This gives the entire left hand side of that equation a more understandable meaning. $\nabla_{\vec{\theta}} \mathbb{E}_{\vec{\epsilon} \sim \mathcal{N}(\vec{0},I)} \{F(\vec{\theta} + \sigma \vec{\epsilon})\}$ which is the value the right hand side evaluates, can be read as "The gradient with respect to a given vector of weights, for the expected value of a function taken with epsilon being a vector of Gaussian noise, where the function is the objective function applied to the given vector of weights adjusted by Gaussian noise by a certain known factor."

The right hand side of the expression then goes on to demonstrate the computation of this value, which is used to perform the optimisation and learning.

First, let us look at the innermost part of the function. The same scoring function evaluated on Gaussian noise adjusted weights is multiplied by the given Gaussian noise vector. This creates a vector where Gaussian noise has been multiplied by the reward function, and will result in noise that resulted in a decrease in score function having all of its positive shifts weighted negatively, and its negative shifts weighted positively. While if the reward is successful and therefore positive, the vector will keep its existing signage. Either way they will be multiplied by the severity of the success or failure.

An expected value is then taken of this, which represents a collecting and averaging of those clustered black dots in Figure 1 (page 3). to find a location that has the best average positive movement in the score function (gradient ascent).

This point is then adjusted to fall on the outskirts of the searched region by dividing it by the standard deviation, allowing for consistent movement based on the area searched regardless of how extreme or muted the output of the expected value for the computed results of each of the black points.

Of course it is worth noting, that the expected value is based on an infinite possible set of Gaussian distribution size appropriate vectors. This cannot be fully computed through existing computation methods, and so is explored through random sampling which results in the cluster of black dots in 1. In the above where I speak of the black dots in the above it would be more appropriate to consider them a black circle with a radius proportional to the set standard deviation.

3 Evolutionary Strategy Algorithm

The algorithms as presented in the original paper are presented in a format more congruent with mathematical expectations than with programming. Following is a restatement of the algorithms in a fashion that is much more consistent with modern programming algorithm descriptions and that should be much more implementable.

The general format of the evolution strategy taken is shown in Algorithm 1 (page 7), which is derived from Equation 1 (page 5).

Algorithm 1: Evolution Algorithm

```
Input: number of generations $g$, number of samples per generation $s$, vector of
       initial weights $\vec{\theta}^0$, noise standard deviation $\sigma$, objective function $F$,
       learning rate $\alpha$
for $t \leftarrow 0$ to $g$ do
    $\vec{\Delta} \leftarrow \vec{0}$
    repeat $s$ times
        $\vec{\epsilon} \sim \mathcal{N}(\vec{0}, I)$
        $\vec{\Delta} \leftarrow \vec{\Delta} + (F(\vec{\theta}^t + \sigma\vec{\epsilon}))\vec{\epsilon}$
    end
    $\vec{\theta}^{t+1} \leftarrow \vec{\theta}^t + \frac{\alpha}{s\sigma}\vec{\Delta}$
end
Output: $\vec{\theta}^{g+1}$
```

The output of Algorithm 1 (page 7) will be a set of weights that represents a local maximum in the possible attainable rewards given by evaluating the network after g generations.

This can be parallelised as shown in Algorithm 2 (page 8) by generating workers with known random seeds and known initial parameters, and instead of doing a loop for each child in a population, instead sending evaluating a worker for each one and synchronising the returned rewards value scalars. After this, each worker can evaluate its new network state through knowledge of the initial random state of every other worker, and the scalar return values that were synchronised.

Algorithm 2: Parallel Evolution Algorithm

```
Input: number of generations $g$, number of samples per generation $s$, vector of
       initial weights $\vec{\theta}^0$, number of weights $n$, noise standard deviation $\sigma$,
       objective function $F$, learning rate $\alpha$, vector of seeds for workers $\vec{\Xi}$
for $t \leftarrow 1$ to $s$ do
    initialise worker $\vec{W}_t$ with seed $\vec{\Xi}_t$
end
for $t \leftarrow 0$ to $g$ do
    for $i \leftarrow 1$ to $s$ do
        on worker $\vec{W}_i$ begin
            $\vec{\epsilon} \sim \mathcal{N}(\vec{0}, I)$
            global $\vec{r}_i \leftarrow F(\vec{\theta}^t + \sigma\vec{\epsilon})$
        end
    end
    for $i \leftarrow 1$ to $s$ do
        await worker $\vec{W}_i$
    end
    for $i \leftarrow 1$ to $s$ do
        on worker $\vec{W}_i$ begin
            $\vec{\Delta} \leftarrow \vec{0}$
            for $j \leftarrow 1$ to $s$ do
                $\vec{\epsilon} \sim \mathcal{N}(\vec{0}, 1)$ with seed $\vec{\Xi}_j$ on access $t \times n$
                $\vec{\Delta} \leftarrow \vec{\Delta} + (\vec{r}_j)\vec{\epsilon}$
            end
            $\vec{\theta}^{t+1} \leftarrow \vec{\theta}^t + \frac{\alpha}{s\sigma}\vec{\Delta}$
        end
    end
end
Output: $\vec{\theta}^{g+1}$
```

4 Results

Two standard experiments were used to test the ES system that was provided, and the paper and associated blog post found the results competitive with traditional reinforcement learning based approaches.

The first of these was MuJoCo, or Multi-Join dynamics with Contact. The environment is a set of angles that each joint in an agent is in, the expected outputs of the network are the amount of torque that should be applied to each joint, the reward function is based on how far forward the agent has moved.

TABLE 1 MuJoCo Results, with TRPO and ES performance given in timesteps.

Environment	%	Score	TRPO	ES	ES / TRPO
HalfCheetah	25%	-1.35	9.05E+05	1.36E+05	0.15
	50%	793.55	1.70E+06	8.28E+05	0.49
	75%	1589.83	3.34E+06	1.42E+06	0.43
	100%	2385.79	5.00E+06	2.88E+06	0.58
Hopper	25%	877.45	7.29E+05	3.83E+05	0.53
	50%	1718.16	1.03E+06	3.73E+06	3.62
	75%	2561.11	1.59E+06	9.63E+06	6.06
	100%	3403.46	4.56E+06	3.16E+07	6.93
InvertedDoublePendulum	25%	2358.98	8.73E+05	3.98E+05	0.46
	50%	4609.68	9.65E+05	4.66E+05	0.48
	75%	6874.03	1.07E+06	5.30E+05	0.50
	100%	9104.07	4.39E+06	5.39E+06	1.23
InvertedPendulum	25%	276.59	2.21E+05	6.25E+04	0.28
	50%	519.15	2.73E+05	1.43E+05	0.52
	75%	753.17	3.25E+05	2.55E+05	0.78
	100%	1000.00	5.17E+05	4.55E+05	0.88
Swimmer	25%	41.97	1.04E+06	5.88E+05	0.57
	50%	70.73	1.82E+06	8.52E+05	0.47
	75%	99.68	2.33E+06	1.23E+06	0.53
	100%	128.25	4.59E+06	1.39E+06	0.30
Walker2d	25%	957.68	1.55E+06	6.43E+05	0.41
	50%	1916.48	2.27E+06	1.29E+07	5.68
	75%	2872.81	2.89E+06	2.31E+07	7.99
	100%	3830.03	4.81E+06	3.79E+07	7.88

This found that in general, ES reached certain scores less quickly than traditional reinforcement learning (trust region policy optimisation), though no worse than a factor of 10. The full results for this can be seen in Table 1 (page 9).

The paper and blog post highlight however, that timesteps are perhaps not the greatest measure of efficiency, considering that the major speed up gained by using ES would come from parallelisation and therefore real execution time, not by pure number of timesteps evaluated.

When looking at the most complex MuJoCo task, a 3D Humanoid, it was possible to solve with ES by using 1440 CPUs across 80 machines in only 10 minutes. Compared with 32 A3C workers on one machine which take about 10 hours. This is because of the high communication bandwidth that is required for parallel reinforcement learning, which is not required by Algorithm 2 (page 8).

Alongside this, across 720 cores in 1 hour was comparable to A3C 32 cores in 1 day. The full results of the Atari can be seen in Table 2 (page 11). Again this takes advantage of the trivialised parallelisation and small size of synchronised data to perform large scale distribution of the problem that would not otherwise be easily possible with the other AI techniques it is compared with.

TABLE 2 Atari Games Comparison of Different AI Techniques

Game	DQN	A3C FF, 1 day	HyperNEAT	ES FF, 1 hour	A2C FF
Amidar	133.4	283.9	184.4	112.0	548.2
Assault	3332.3	3746.1	912.6	1673.9	2026.6
Asterix	124.5	6723.0	2340.0	1440.0	3779.7
Asteroids	697.1	3009.4	1694.0	1562.0	1733.4
Atlantis	76108.0	772392.0	61260.0	1267410.0	2872644.8
Bank Heist	176.3	946.0	214.0	225.0	724.1
Battle Zone	17560.0	11340.0	36200.0	16600.0	8406.2
Beam Rider	8672.4	13235.9	1412.8	744.0	4438.9
Berzerk	0.0	1433.4	1394.0	686.0	720.6
Bowling	41.2	36.2	135.8	30.0	28.9
Boxing	25.8	33.7	16.4	49.8	95.8
Breakout	303.9	551.6	2.8	9.5	368.5
Centipede	3773.1	3306.5	25275.2	7783.9	2773.3
Chopper Command	3046.0	4669.0	3960.0	3710.0	1700.0
Crazy Climber	50992.0	101624.0	0.0	26430.0	100034.4
Demon Attack	12835.2	84997.5	14620.0	1166.5	23657.7
Double Dunk	21.6	0.1	2.0	0.2	3.2
Enduro	475.6	82.2	93.6	95.0	0.0
Fishing Derby	2.3	13.6	49.8	49.0	33.9
Freeway	25.8	0.1	29.0	31.0	0.0
Frostbite	157.4	180.1	2260.0	370.0	266.6
Gopher	2731.8	8442.8	364.0	582.0	6266.2
Gravitar	216.5	269.5	370.0	805.0	256.2
Ice Hockey	3.8	4.7	10.6	4.1	4.9
Kangaroo	2696.0	106.0	800.0	11200.0	1357.6
Krull	3864.0	8066.6	12601.4	8647.2	6411.5
Montezumas Revenge	50.0	53.0	0.0	0.0	0.0
Name This Game	5439.9	5614.0	6742.0	4503.0	5532.8
Phoenix	0.0	28181.8	1762.0	4041.0	14104.7
Pit Fall	0.0	123.0	0.0	0.0	8.2
Pong	16.2	11.4	17.4	21.0	20.8
Private Eye	298.2	194.4	10747.4	100.0	100.0
Q*Bert	4589.8	13752.3	695.0	147.5	15758.6
River Raid	4065.3	10001.2	2616.0	5009.0	9856.9
Road Runner	9264.0	31769.0	3220.0	16590.0	33846.9
Robotank	58.5	2.3	43.8	11.9	2.2
Seaquest	2793.9	2300.2	716.0	1390.0	1763.7
Skiing	0.0	13700.0	7983.6	15442.5	15245.8
Solaris	0.0	1884.8	160.0	2090.0	2265.0
Space Invaders	1449.7	2214.7	1251.0	678.5	951.9
Star Gunner	34081.0	64393.0	2720.0	1470.0	40065.6
Tennis	2.3	10.2	0.0	4.5	11.2
Time Pilot	5640.0	5825.0	7340.0	4970.0	4637.5
Tutankham	32.4	26.1	23.6	130.3	194.3
Up and Down	3311.3	54525.4	43734.0	67974.0	75785.9
Venture	54.0	19.0	0.0	760.0	0.0
Video Pinball	20228.1	185852.6	0.0	22834.8	46470.1
Wizard of Wor	246.0	5278.0	3360.0	3480.0	1587.5
Yars Revenge	0.0	7270.8	24096.4	16401.7	8963.5
Zaxxon	831.0	2659.0	3000.0	6380.0	5.6

5 Conclusions

On a surface per-timestep of evaluation level, evolution strategies perform worse than reinforcement learning techniques. The benefit they have is their ease of large scale parallelisation with only a single scalar value requiring cross communication from each worker. This results in an extremely scale efficient model that is not effective with other techniques. Diminishing returns mar most reinforcement learning techniques and prevent large scale multiprocessing, however evolution strategies using the model laid out in Algorithm 2 (page 8) do not suffer from this problem.

It is worth noting that there are however still some diminishing returns, as more additional processing power was given to ES than was gained in computation speed up, however these are vastly reduced when compared with parallel backpropagation.

There is one noteworthy downside with the effectiveness of this ES technique, and that is that is does not perform well on supervised learning problems where it is possible to compute the exact gradient of the loss function with backpropagation. This represents a poor performance on tasks such as image classification, speech recognition, or any other broad categorisation and recognition task. In this respect, ES is far more effective at dealing in learning how to interface with complex systems (robotics, gaming) than it is with the generally lower complexity categorisation and identification problems.

One other aspect of this technique that is not as readily focused on is ease of understanding and development. The technique is incredibly easy to understand and implement, and can be taught with little requiring at its core only a basic understanding of vectors and probability. The reason the approach is effective at locating local maximums if intuitive, and the entire system functions in a much more transparent fashion than backpropagation.

This gives ES a gravitas that is not focused on in the report, blog, or any surrounding material. With ease of implementation comes a reduced opportunity for bugs or failures. Implementing neural network training algorithms can be exceptionally difficult, and given the black-box nature of the programs: very difficult to debug. It is possible to misconfigure the network and teaching algorithm and still have it learn and train at a much worse rate simply by chance that the neurons can still establish some sensible notion of a local maximum, gradient ascent, and connectivity. Tracking down bugs in such as system is exceptionally difficult as all might simply appear to work with reduced efficiency from and outside perspective.

References

[1] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[2] A. Karpathy, T. Salimans, J. Ho, P. Chen, I. Sutskever, J. Schulman, G. Brockman, and S. Sidor, "Evolution strategies as a scalable alternative to reinforcement learning," Mar 2019. [Online]. Available: https://openai.com/blog/evolution-strategies/

[3] Z. Huo, B. Gu, Q. Yang, and H. Huang, "Decoupled parallel backpropagation with convergence guarantee," *arXiv preprint arXiv:1804.10574*, 2018.

[4] D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber, "Natural evolution strategies," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 3381–3387.

[5] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural evolution strategies," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 949–980, 2014.

[6] S. Yi, D. Wierstra, T. Schaul, and J. Schmidhuber, "Stochastic search using the natural gradient," in *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 2009, pp. 1161–1168.

[7] T. Glasmachers, T. Schaul, and J. Schmidhuber, "A natural evolution strategy for multi-objective optimization," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2010, pp. 627–636.

[8] T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber, "Exponential natural evolution strategies," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 393–400.

[9] T. Schaul, T. Glasmachers, and J. Schmidhuber, "High dimensions and heavy tails for natural evolution strategies," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 845–852.