Evolutionary Algorithms for Reinforcement Learning

Project for Reinforcement Learning

RAPPAPORT Gabrielle
gabrielle.rappaport@student.ecp.fr
BEN-GOUMI Meryem
meryem.ben-goumi@student.ecp.fr
LAZRAQ Yahya
yahya.lazraq@student.ecp.fr
MARSAL Romain
romain.marsal@student.ecp.fr

March 29, 2019

Abstract

This report provides an overview of the paper on *Evolutionary Algorithms for Reinforcement Learning* by David E. Moriaty, Alan C. Schultz, and John J. Grenfenstette (link). The document starts with a reminder of Reinforcement Learning principles, followed by an explanation of the specificity and the functioning of Evolutionary Algorithms. Next, we present two different implementations using Evolutionary Algorithms for Reinforcement Learning problems. The first one illustrates the "genetic" algorithm, and the second one implements the EA for Deep Reinforcement Learning in the context of the CartPole game. The examples allows us to conclude on the strengths and weakness of EARL, that we exposed in the last section. The codes associated to the two examples are available on the following link.

1 Presentation of Reinforcement Learning

The main goal of Reinforcement Learning algorithms is to help find the best sequence decision for an agent to reach a defined goal through experiments and interactions with the environment. The state of an agent in the environment is represented by $s_t = \delta(s_{t-1}, a_{t-1})$ with s_t being the state at time t and a_t being the action taken by the agent at time t. From the state s_t, taking the action a_t could lead to a reward $r(s_t)$.
The goal of the agent is to learn the function $\pi : S \to A$ that associates to each state an action, called the policy function. The idea for the agent is to know each action to take at time t knowing that he is in state s_t. We can then define $V^{\pi}(s)$, which corresponds to the cumulative rewards that will be obtained starting from state s. Therefore, the optimal policy π^* is defined as the one maximizing the cumulative reward for each state s:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \tag{1}$$

Two approach are then possible. The first one is to try to find directly the optimal π^* in the policy space, and the other one implies to look for the optimal V^{π^*} in the value function space.
The value function methods try to maximize the cumulative rewards without explicitly determining the policy. The value function is then learned through experiments. The most common method to do so is called the Temporal Difference method. We obtain then the optimal action through:

$$\pi(s) = \arg\max_{a} V^{\pi^*}(\delta(s, a)) \tag{2}$$

The problem is that $\delta(s, a)$, called the transition function is unknown. Hence, we define the Q-function by:

$$Q(s, a) = r(s) + V^{\pi}(\delta(s, a)) \tag{3}$$

This Q-function is learned through Temporal difference and then we could get π^* thanks to:

$$\pi(s) = \arg\max_{a} Q(s, a) \tag{4}$$

The other method for reinforcement learning is to search on the policy-space and to get to directly know the optimal policy function. Different methods are used to do that, and evolutionary algorithms is one of them.

2 Evolutionary Algorithm for Reinforcement Learning

The Evolutionary Algorithms (EAs) are inspired from the theory of evolution through natural selection. EAs are a search technique applied in various areas as combinatorial optimization. We will focus for this report on the EA applied to RL.
The idea behind the EA is to update a pool of good solutions through basic genetic modification operations (mutation, recombination ...) to get better solutions. The evaluation is

done thanks to a fitness function. A solution is represented by structures called chromosomes. Subtracts of the solution, called genes, are modified in each iteration (or generation). Agents in each iteration having the best fitness score are more likely to impact next generation (as on the natural selection). For RL, we then need to fix two elements:

1. Which representation to map the policy space into chromosomes?
2. Which fitness function to evaluate each population?

Getting a good fitness function helps focus the searches on strong performance elements of the policy space. They fitness function needs to translate the accumulated rewards taken by the agent, but could also reflect the amount of delay.

Regarding the mapping of the policy space into chromosomes, even if there is no limit to policy space representation, two methods are mainly used:

1. Single-Chromosome representation of policies: the agent decision policy is represented as a single genetic representation (chromosome) changing through iterations.
2. Distributed representation of policies: the decision policy is split among smaller elements.

On the next sections of the report, we will focus on the single genetic representation. Indeed, the FrozenLake Problem uses a rule based policy, while the Implementation of CarPole involves a neural net representation of policies.

3 Simple EARL : Single chromosome per policy

In this section we will develop the "single chromosome algorithm", also called "genetic" algorithm, which is a family of algorithms inspired by biological evolution. These methods are easily understandable and may find good solutions for optimization problems but do not guarantee to find the global solution (contrary to the Deep Reinforcement Learning that we will evoke in the next section).
The basis of single chromosome per policy algorithm is to represent its solution as a string of chromosomes (an array of integers, each integer representing an action on the environment) and requires also defining a fitness function to evaluate the solution.
The single chromosome algorithm works by initializing random "chromosomes" and making them evolve in the environment. Iteratively, the chromosomes performing well according to the fitness score will mutate easily in the next generation through specific mutation processes (crossover, mutation etc). Similarly to species evolution, the (k+1)th generation is created from the kth generation that consists of 3 steps.

- Selection Select the sample of best-performing chromosomes from the current generation and set them as parent. They will be the most likely survivors.
- Crossover Mix randomly alleles from the selected parents to generate new offspring child solution.

- MUTATION Transform with a given probability one gene of the selected parent into a new random allele.

3.1 Implementation of FrozenLake Problem

The main limitation of the genetic algorithm described above is that each integers must represent one action to perform on a given Environment. That's why we will use a simple example from the OpenAI gym.

3.1.1 Environment

The agent controls the movement of a character in a grid world. Some cells of the grid are walkable, and others lead our agent into the water. Moreover, the movement of the agent is not perfectly determined by the actions but is also uncertain due to the slippery grid. We define the 4 possible states below:

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

Figure 1: 4 possible States in the Environment

With the mathematical notation:

$$s = (S, F, H, G), \text{ s } \in S \tag{5}$$

The episode is over when the agent falls into the water or when the agent finds the frisbee.

$$S^* = \{s \in S \ / \ s = H \text{ or } s = G\} \tag{6}$$

At each step, you get a reward of 0 except when you reach the goal you will get a $R = 1$.

3.1.2 Genetic algorithm

In the Frozen Lake Problem, each chromosome will have a size 16 (as the number of cells on the grids) with each dimensions taking its value between 0 and 3 included. We initialize our algorithm with a population of 400 and we iterate through 20 epochs (or generation) with the 3 steps mentioned above. The following figure describes the procedure.

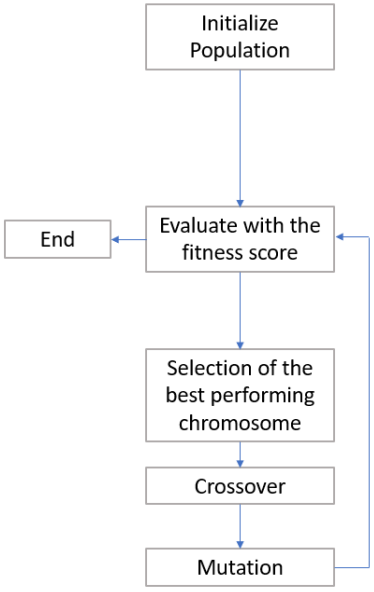

Figure 2: Flochart of the genetic algorithm

3.1.3 Results

Although the genetic algorithm is very simple, it still performs 3 times better than a Random search with a score of 0.85 and 0.3 respectively. In order to optimize both memory space and performance, we analyzed the influence of hyperparameters on the performance. Particularly, we looked at the probability of mutation (p_m), the number of agents per generation (n_a) and the number of generation (n_g).

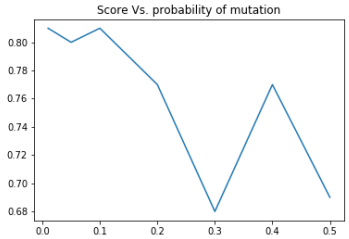

Figure 3: Prob. of mutation

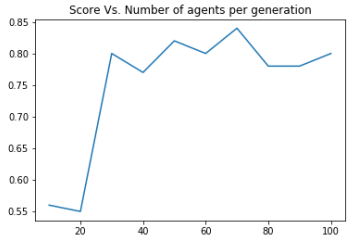

Figure 4: Agents per gen.

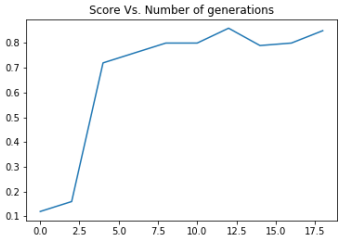

Figure 5: Number of gen.

There is no clear trend for the influence of p_m, that's why we will set it as low as possible. For n_a, the optimal value seems to be around 30. It is not useful to take more penguins exploring our environment because the score won't increase. Finally, the optimal value for n_g is 11, why means our 30 agents will mutate over 11 generations to give a population able to perform well on the Frozen Lake

4 Second Implementation: EA for Deep RL

In this section, we validate the effectiveness of evolutionary algorithms and prove that genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning.
The basis of EA for Deep Reinforcement Learning is to train neural networks modifying the neural network parameters accordingly to genetic behavior, instead of using Reinforcement Learning coupled with traditional gradient back-propagation.
This EA for Deep Reinforcement Learning approach consists in generating a distribution of parameters for the neural network and in having a large number of agents operating in parallel, using parameters sampled from this initial distribution. Each agent acts in his own environment so as to compute the reward, ie the fitness score of the agent in the environment. With this score, it is possible to select the best performing agents and to generate new parameter distributions from theirs. Thus, we move away from the distributions of inefficient agents and we get closer to the parameters of the most powerful agents, while exploring their neighboring configurations. By repeating this approach many times, with hundreds of agents, the weight distribution shifts to a space that provides agents with a good policy to solve the task at hand.

4.1 Implementation of CarPole

4.1.1 Environment

So as to demonstrate how Evolutionary Algorithms are applied to Reinforcement Learning, we will look closely at the classic example of CarPole, available in OpenAI Gym, in which the goal is to balance a pole connected with one joint on top of a moving cart.

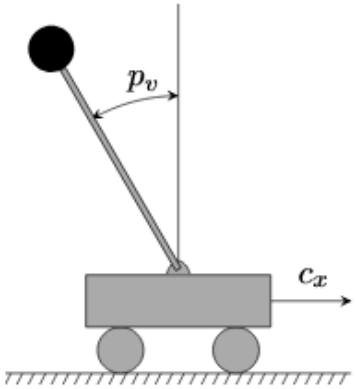

Figure 6: Illustration of the CarPole Problem (source: Wikipedia)

In the Carpole problem, also called the reserved pendulum problem, a pole (or pendulum) is attached to a cart with an un-actuated joint. The Cart moves horizontally on a frictionless path, and is controlled through a force which value is equal to +1 or -1. The initial position

of the cart is upright, and the objective of the agent is to keep the pole in equilibrium while the cart moves. The State Space is composed of four types of values: the position and speed of the cart along the horizontal axis, as well as the angle and speed of the pole.

$$s = (c_x, \dot{c_x}, p_v, \dot{p_v}), \text{ s } \in S \tag{7}$$

The episode is over when the angle of the pole surpasses 15 degrees from vertical, or when the cart moves further than 2.4 units from the center. The resulting absorbing state is thus:

$$S^* = \{s \in S \ / \ |c_x| \geq 2.4 \text{ and } |p_v| \geq 25\} \tag{8}$$

A reward of +1 is provided for every state observation in which the pole remains upright. As a quick reminder of how evolutionary algorithms are implemented for reinforcement learning, we have the following :

- **The policy** corresponds to the parameters, ie the genes that specify the agent behavior. In the context of Deep reinforcement learning, the parameters or genes will be the parameters of our neural network.
- **The rewards** indicate the agent fitness to the environment.

4.1.2 General Behavior of the algorithm

You will find the global behaviour of the EA algorithm on Figure 7, that we will detail on the following paragraphs.

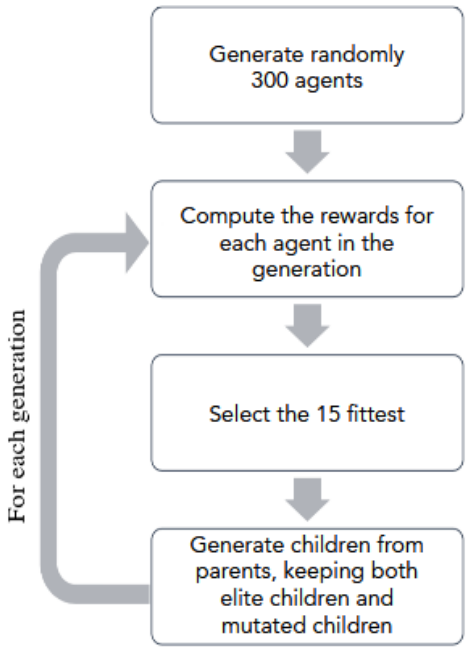

Figure 7: General Behavior of the algorithm

4.1.3 Generations

In the first generation, we generate 300 parents with random parameters. Once generated, we evaluate the performance of each agent by computing the mean of their rewards after three runs so as to minimize luck for each agent. Some of those agents perform better than others and the 15 fittest agents are selected to create the next generation.

4.1.4 Mutations

To produce the next generation, we replicate the 15 selected agents, and mutate the genes by applying additive Gaussian noise to all parameters while copying an agent. That way, we get to explore the neighborhood around parameters of best agents.
We also keep the top performing agent as is, without adding noise, so that we always preserve the best agents as insurance against Gaussian noise causing a probable reduction in performance.

4.1.5 Formalization

Evolutionary algorithm implementations store each individual as a parameter vector , such as θ_n is an offspring of θ_{n-1}.
We can thus describe our model with the following equation and with the following figure.

$$\theta_n = \Psi(\theta_{n-1}, \tau_n) = \theta_n + \sigma * \epsilon(\tau_n)$$

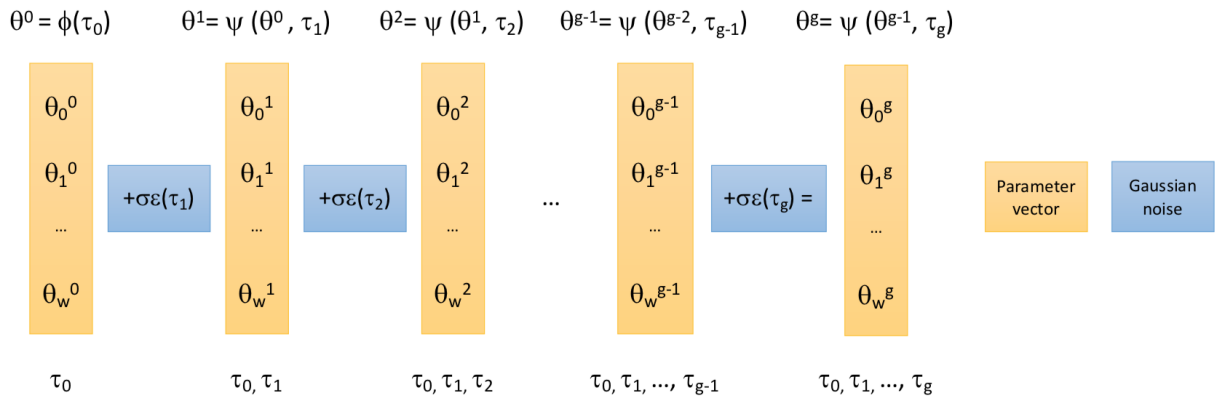

Figure 8: Visual representation of the encoding method

In the previous, we have :

- θ_n an offspring of θ_{n-1}
- $\Psi(\theta_{n-1}, \tau_n)$ a deterministic mutation function
- τ the encoding of θ_n consisting of a list of mutation seeds

- $\theta_0 = \phi(\tau_0)$, where ϕ is a deterministic initialization function
- $\epsilon(\tau_n)\ N(0,1)$, a Gaussian function used to create noise

From a randomly initialized parameter vector θ_0 (produced by an initialization function ϕ seeded by τ_0), the mutation function Ψ (seeded by τ_1) applies a mutation that results in θ_1. The final parameter vector θ_g is the result of a series of such mutations. Recreating θ_g can be done by applying the mutation steps in the same order. Thus, knowing the series of seeds $\tau_0 ... \tau_g$ that produced this series of mutations is enough information to reconstruct θ_g (the initialization and mutation functions are deterministic). Since each τ is small, and the number of generations is low (order hundreds or thousands), a large neural network parameter vector can be stored compactly.

5 Strengths and Weaknesses of EARL

5.1 Strengths of EARL

5.1.1 Storage and Computational Time for large State Spaces

Generally, Deep Neural Network are trained with back-propagation algorithms, that are gradient-based. This training strategy is used to solve Reinforcement Learning problems, when performing searches in Value function space or in Policy space. However, such gradient-based methods are challenging in terms of computational time, and are hence unsuitable for Large State Spaces. In fact, hard deep RL problems hold a large number of state-action pairs, making it impossible to rely on methods that require computing and storing statistical values for each state-action pair.

It turns out that Deep Neuroevolution allows to overcome this dimensionality barrier. EARL policy representations have *generalization* and a *selectivity* aspects that reduce computational time and storage required:

- **Generalization**: in EARL, the policies are specified at a higher level of abstraction than an upfront mapping between observed states and actions. Still, the level of generality in the definition of the policy varies depending on the EARL policy representation chosen. In particular, when evolving DNNs, the policy is specified implicitly within the weights (connection and biais weights) of the Network. For example, in the implemented example of CarPole game, there is no need to store the policy for each state-action pair, as it is implied by θ, the vector containing the weights of the neural net.
- **Selectivity**: since EA algorithms are based on the idea of evolution by Natural Selection, the policies for EARL are selectively represented. In fact, the fittest policies are kept in memory, whereas the ones leading to worse decisions are omitted. In the CarPole game example, only the mappings of the top 15 fittest agents are learned.

Hence, EARL systems make it possible to scale up to realistic RL problems with large state spaces. Indeed, researchers from Uber AI Labs claim that: *"(it) is fast, enabling*

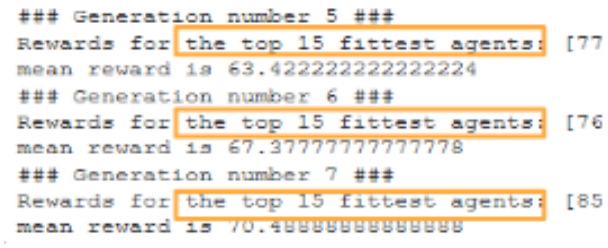

Figure 9: Screen Shot of the Output when running different generations

training Atari in 4 *hours on a single desktop or* 1 *hour distributed on 720 CPUs."* [2]. For a fair comparison, the same paper states that training Atari on a single desktop takes $7 - 10$ days for a Deep Q-Learning algorithm, and 4 days when using policy gradients.

5.1.2 Performance for Ambiguous State Observations

In realistic RL problems, sensors might not be able to completely describe the observed states. This missing information can mislead strategies based on Temporal Difference, since they directly match rewards with singular decisions. Hence, when two states are impossible to distinguish, the matching is mistaken because it combines the rewards from the two ambiguous states. On the contrary, EARL systems are more robust towards this *Hidden State* problem because they rely on the final output of a series of decisions, instead of relying on individual decisions and sensor information.

5.1.3 Ability to adapt to non-stationary environments

Another challenge posed by some difficult RL problems is the evolution of the environment over time. For Temporal Difference methods, this issue is addressed with an appropriate trade-off between *exploration* and *exploitation*. On the other hand, it is interesting to note that EARL systems are generally able to keep track of a non-stationary fitness function, assuming that the population remains enough diversified over time and that the environment's changing pace is moderate compared to the computational time required for policy evaluations. For example, our implementation of the EA algorithm, that we applied to the CarPole game, can be replicated for a non-stationary environment. Such replication might require a higher mutation rate, so that the population is enough diversified to respond to environment variations. EARL systems that rely on distributed policy representations are particularly compatible with this type of problem. In fact, such policy representations are based on the idea according to which the optimal solution is obtained when compounding several individual solutions. Since no individual agent can offer the optimal solution, the population remains diverse, and can hence react to changes in the environment. The SANE (Symbiotic, Adaptive Neuro-Evolution) system is an example of method using distributed policy representations. It is well suited for non-stationary environments because of its notable ability to maintain a diversified population and to search for a combination of partial solutions.

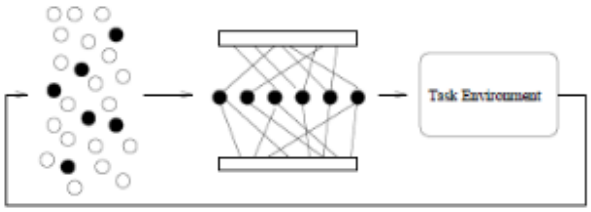

Figure 10: Representation of the SANE system. Source:[3]

5.1.4 Weaknesses of EA for Deep Reinforcement Learning

It should be noted that there are a few challenges to the application of Evolutionary Algorithms to Deep Reinforcement Learning. The main difficulty is related to the *Mutation* process: there is no clear guidance on how the mutations should be performed, whilst they have a direct impact on the learning process and hence on the final performance. In the example of CarPole game, the genes are mutated by adding Gaussian noise to the parameters of selected agents. However, there is no rule that helps monitor the multiplicative factor used, called `mutation_power` in our function `mutate`. The value that was set, 0.2, is the one suggested in reference [2]. Yet, it seems that such value was obtained empirically, after testing different values and choosing the one that works best.
Another challenge is the direct consequence of the *Selectivity* aspect of EARL methods, mentioned above as a strength. Indeed, policies that do not perform well are eliminated, which reduces storage and computational time. However, this selection of information can reduce the performance with regards to **rare states**. Indeed, when a state is scarcely visited, the corresponding choice of "best" action is described by a gene that might be altered by mutations. This can be caused by an accumulation of mutations in the gene: the values become random because they don't directly impact on the fitness result of the agent.

6 Conclusion

In conclusion, reviewing the reference paper has allowed us to understand the concept of Evolutionary Algorithms and its application to Reinforcement Learning Problems. EARL seems to indeed be a powerful alternative to Temporal-Difference methods. As a next step, it would be interesting to understand why, in some specific Reinforcement Learning problems, the algorithm fails. Reference [2] offers interesting insights to further understand this issue when evolving Deep Neural Networks.

References

[1] The paper in which this report is based, *Evolutionary Algorithms for Reinforcement Learning* by David E. Moriaty, Alan C. Schultz, and John J. Grenfenstette

[2] Deep Neuroevolution, *Genetic Algorithms are a Competitive Alternative forTraining Deep Neural Networks for Reinforcement Learning*, Felipe Petroski Such Vashisht Madhavan Edoardo Conti Joel Lehman Kenneth O. Stanley Jeff Clune 2018.

[3] Adaptive Behavior, *Incremental Evolution of Complex General Behavior*, Faustino Gomez, Risto Miikkulainen 1996.

[4] A gist by Moustafa Alzantot, on which our implementation of Frozen Lake is based. Link.

[5] A blog-post by Paras Chopra on which our implementation of CarPole is based. Link.