Genetic Strategies or Reinforcement Learning?

Mátyás Vincze, Mostafa Haggag, Sara Miotto

***Abstract*—Twin Delayed Deep Deterministic Policy Gradients, Covariance Matrix Adaptation Evolution Strategy, NeuroEvolution of Augmenting Topologies and Genetic Algorithms, are all widespread in literature. In this project we want to make a comparison between them in the Bipedal Walker environment from OpenAI Gym. We will find out that there is not a proper winner.**

Index Terms— Reinforcement Learning, TD3, Evolutionary algorithms, NEAT, Genetic Algorithm, Feed Forward Neural network, Recurrent Neural network, Open-AI Gym, Biped Walker.

I. Introduction

REINFORCEMENT learning, RL, is the study of the agents and of the way they learn by trial and error. It formalizes the idea that rewarding as well as punishing an agent for its behavior, makes it more likely to repeat or forego that behavior in the future. In particular, as a RL technique, we have chosen to implement Twin Delayed Deep Deterministic Policy Gradient, as it has proven to be a stable solution for the environment we have tried to solve. Essentially what RL tries to do is to replicate the learning cycle of organisms over their lifetime and this process of learning goes on until an optimal agent is created.

An alternative method to RF was introduced by Karpathy in [1]. It deals with the employment of evolutionary strategies as a way to find the best combination of weights and so to achieve the highest possible reward. For this purpose we have tried both a genetic algorithm [2] and an evolutionary strategy, CMA-ES [3]. Genetic Algorithms, GAs, and Evolutionary Strategies, ES, are very similar. Evolutionary strategies take inspiration from the evolution of the species, where randomness guides exploration and only individuals with positive mutations can survive. Genetic Algorithm is a metaheuristic search inspired by the theory of natural evolution theorised by Charles Darwin. It reflects the process of natural selection where the fittest individuals are selected for reproduction. While GAs and ES deal with the variation of weights, Neat, [4], is an algorithm concerned with the topology of a neural network.

The sections of the paper are structured as follows. Firstly, the Methodologies section will specify the environment used to set up all the experiments, as well as the four different techniques undertaken for this scope. Secondly, we have a chapter Results where graphs and statistics are reported so as to be able to analyze the results. Then we have a chapter devoted to the difficulties encountered throughout the whole project and one committed to the lessons learned. Finally, the conclusion summarizes everything.

II. Methodologies

In this project, we have tested the previously chosen algorithms in the Bipedal Walker environment. The programming language employed was Python.

Bipedal walker environment is part of the Open AI Gym library, a toolkit that provides a wide variety of simulated environments. We have opted for it as it seemed very suitable for training and comparing agents and particularly Bipedal Walker seemed to offer rich dynamics. The robot at issue has 2 legs, each one with 2 joints. He learns to walk by applying torque on his joints. The dynamics of his movements are deterministic but the terrain is randomly generated at the beginning of each episode. The environment has both continuous action and observation space. The task is to move the robot forward as much as possible. However, some difficulties encountered by bipedal walkers are due to the non-linearity of the environment (the behavior of the robot changes according to the different situations), the uncertainty (terrain varies), the sparsity of the rewards and the partial observability (the robot can look ahead with lidar measurements but not behind). The agent gets a positive reward proportional to the distance walked on the terrain. It can get a total of +300 rewards all the way up to the end. If the agent tumbles, it gets a reward of -100.

We have also tried to use both feed-forward and recurrent neural networks, but, as expected, the performances with recurrent were quite worse than the ones obtained with feed-forward. This was due to the fact that we were not dealing with a sequential problem where history could be used as effectively as in normal sequential problems. So, recurrent connections tended to over-complicate the solution.

A. Genetic Algorithm

Artificial neural networks can use a genetic algorithm [2] with the purpose of optimizing a fitness function. A genetic algorithm evolves a population P of N individuals (here, neural network parameter vectors θ, called genotype). At every generation, each θ_i is evaluated, producing a fitness score $F(\theta_i)$.

First and foremost, it was necessary for us to turn the weights and the biases of the neural network into the genotype. So matrices were flattened and bias vectors were concatenated to them. After one episode the genotype vector was converted into the input neural network architecture. After this preparation phase, the modus operandi of the genetic algorithm covered all the classical steps: an initial random population, a fitness function (the cumulative reward assigned to the biped),

a way of selecting parents (roulette-wheel selection), replacement of generations implemented as elitism (we kept every time the four best individuals among parents and children), mutations and cross-over.
As far as mutations were concerned, they were applied by the supplement of an additive Gaussian noise to the parameter vector: $\theta' = \theta + \sigma\epsilon$ where $\epsilon \Rightarrow N(0, I)$.
Data were analyzed by monitoring the performances and all these steps were repeated until the maximum number of generations have been met. At the end of the code, there is also a hint towards the technique of the Student-Teacher method from unsupervised domain adaptation. In this case it is used so as to train a new neural network by comparing it with a reference one and measuring the relative loss (Mean Squared Error).

B. Neat

Neat [4] is a classical example of neuroevolution which aims at creating artificial neural networks. The basic idea is to try to develop the structure of a neural network by means of a genetic algorithm rather than using a fixed structure. Complexification of a minimal structure is one of the most important concepts. In fact, the search begins in a minimal-topology space as lower-dimensional structures are more easily optimized, and at the beginning, we have no hidden nodes. Each individual in the initial population is simply made of input nodes, output nodes, and a series of connection genes (weights) between them. Direct encoding is applied to a population of individual genomes. Each genome contains two sets of genes: node genes, which specify single neurons, and connection genes, which specify where connections come into and out of, the weights of such connections and whether or not they are enabled. NEAT can count also on the concepts of mutation and crossover. Mutations can be applied to existing connections as well as they can add new ones. Regarding crossover, this algorithm uses a particular strategy known with the name of historical markings. The idea is that we do not like a blind crossover, yet we want our crossover to generate better individuals with a high probability. So, each time a new node or new type of connection occurs, a historical marking is assigned, allowing easy alignment when it comes to breed two of our individuals. In biology, it is the so-called homology. Another peculiar aspect of NEAT is speciation which splits up the population into several species that are similar between each other in topology and connections. In this way, individuals in a population only have to compete with other individuals within that specie. This allows for a new structure to be created and optimized without fear that it will be eliminated before it can be truly explored. The fitness function is always calculated as the cumulative rewards assigned to Bipedal Walker.

C. Covariance Matrix Adaptation Evolution Strategy

The CMA-ES is an evolutionary method that tries to develop the policy by adopting different methods introduced by Hansen [3]. This is done in order to give birth to individuals in a generation with better performance. We have employed the structure of a neural network with weights, θ, and by using CMA algorithm, we have tried to find the best candidate parameters. In each generation G, there is a list of potential parameters, $\theta_i, ...\theta_n$. Each θ_i is sampled from a multivariate Gaussian distribution with zero mean and a covariance matrix Σ (it is rotated and adjusted in order to be adapted to the fitness landscape). The search distribution has a mean value of μ and a standard deviation of σ (step size) as seen in eq.1.

$$\theta_i \sim \mu + \sigma N(0, \Sigma) \tag{1}$$

The standard deviation σ accounts for the level of exploration. During the search, it gathers information on (un)successful mutations, thus building an evolution path. The path is used to adapt both the step-size σ (shrinking or amplifying it depending on successful mutations) and the covariance matrix C (rotating and adjusting it) to adapt the best fitness. The following generation is sampled from the updated distribution.

D. Twin Delayed DDPG

Twin Delayed DDPG (TD3) considers a standard reinforcement learning setup where an agent interacts with an environment E within a discrete time. At each time step t the agent perceives an observation $s_t \in S$, takes an action $a_t \in A$ and obtains a reward r_t. The decision-making of the agent is defined by a policy: $\pi : S \rightarrow P(A)$ and this policy is modeled as a Markov decision process (MDP). The goal is to learn a policy π which maximizes the expected return (sum of discounted future rewards, $R(\tau)$) from the starting distribution. A trajectory τ is a sequence of states and actions $(s_0, a_0, s_1, a_1, \ldots)$. The action-value function $(Q^\pi(s_t, a_t))$ is the expected return after taking an action a_t, being in a state s_t, and following a policy π. Knowing the optimal Q^*, we can determine the best policy able to choose the action a which maximizes Q^* at the current state. The Bellman equation defines the value of the starting point as the expected reward obtained by moving from this starting point, plus the value of wherever you land next:

$$Q^\pi(s, a) = \mathbb{E}_{s' \sim P(\cdot|s,a)}[r(s, a) + \gamma \mathbb{E}_{a' \sim \pi(\cdot|s)}[Q^\pi(s', a')]] \tag{2}$$

Thus the Bellman equation for the optimal action-value function is obtained by choosing the action a' maximizing $Q^\pi(s', a')$. Q-learning aims at learning an approximation of $Q^*(s, a)$ off-policy. This means that it uses past experiences, regardless that the policy used to explore the environment has been changed. It is a greedy algorithm so it chooses the best currently available action and update Q-value for the previous state-action pair. As our state-space was continuous, it was not possible for us to apply Q-learning straight away. Finding the greedy policy required the optimization of a_t at each time step and this procedure was too computationally heavy to be feasible. However, we were able to use the actor-critic (policy-Q) approach. Deep Deterministic Policy Gradient (DDPG) learns a Q function and a policy at the same time. By using both off-policy data and the Bellman equation, it learns the Q-function and uses it to formulate the policy. Suppose the approximator is a neural network $Q_\phi(s, a)$, and that we have collected a set D of examples (s, a, r, s', d), where d indicates

whether state s' is terminal. We can set up a mean-squared Bellman error (MSBE) function which tells us how close is Q_ϕ to the Bellman equation:

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s',d)\sim D}[(Q_\phi(s,a) - \tag{3}$$

$$(r + \gamma(1-d)\max_{a'} Q_\phi(s', a'))^2] \tag{4}$$

In DDPG two tricks are applied: replay buffer and target networks. Replay buffer is a finite set D of historical experiences. The term $r + \gamma(1-d)\max_{a'} Q_\phi(s', a')$ is the target which the Q-function wants to reach, but as both of them depend on the same parameters, we delay its update with respect to the Q-function's. To make DDPG policies explore better, we have added noise to the chosen actions during training. Twin Delayed DDPG (TD3) addresses issues of the DDPG algorithm by implementing three tricks. By using clipped double-Q-learning, it learns two Q-functions instead of just one and employs the smaller one in order to form the targets in MSBE. It then updates the policy (and target networks) less frequently than the Q-function. Finally, target policy smoothing adds noise to the target action. In this way, it makes it harder for the policy to exploit Q-function errors.

III. Results

For evolutionary methods, we have performed 10 test episodes for the best individual in each generation and then averaged the test results in order to represent the performance at those time steps. We were following previous work by Zhang [5].

A. Genetic Algorithm

In order to implement this algorithm, we have used a feed-forward neural network with three hidden layers (we omit the comments on the results found with RNN for the reasons already explained in the beginning). The input nodes have been set to 24, the output nodes to 4, and the hidden ones respectively to 20, 12, and 12. Hyperbolic tangents have been employed as activation functions. The algorithm runs for 2000 generations with a population of 30 individuals. As we can see from Fig.1, the best individual reaches a score of almost 163 gradually, without falling too much. The algorithm then maintains a quiet constant score of 150 until the end of the environment. Studying the simulation, we can notice the way of walking of the robot: front leg outstretched and back leg hooked. Talking about the average best score, we see that we achieve a maximum score of 245. Results seem to be pretty stable.

B. Neat

Also for neat [4] we just show the results for feed-forward. We have used hyperbolic tangent activation functions and enabled elitism. This has allowed the preservation of a number of fit individuals in each species from one generation to another. We have started with 1 hidden layer and set the fitness threshold equal to 230 with a population size of 220. We have used the max fitness criteria during our runs. As seen in the figure Fig.2, the algorithm reaches a fitness of 232

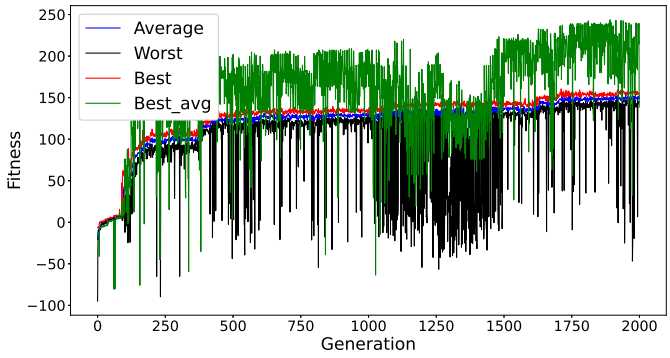

Fig. 1. This is a figure comparing the best, average, worse, and best average fitness scores for each generation for the Genetic algorithm.

in 200 generations and it keeps fluctuating around this value without clear improvements. By looking at the simulation, we can notice that robot has learned how to walk without falling with his back leg hooked and the front leg crawling while maintaining a sort of balance. This is surely not the best way to walk. Furthermore Neat has been shown to have the worst maximum best average best performance with a score of 207 when compared to the others.

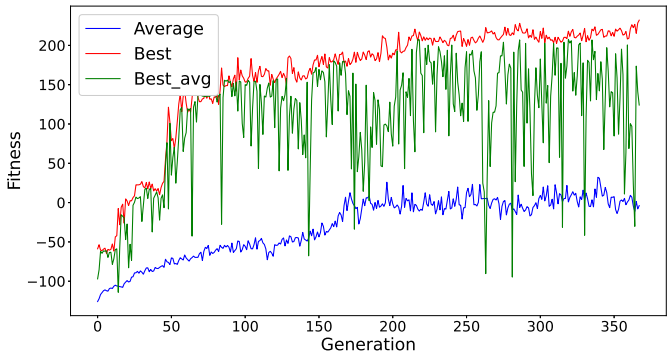

Fig. 2. This is a figure comparing the best, average, worse, and best average fitness scores for each generation for the neat algorithm.

C. CMA-EA

We have used a two-hidden-layer neural network, respectively 30 and 10 nodes, and hyperbolic tangents as activation functions. The initial standard deviation has been set to 0.3 and the runs to 2000 generations, with each generation made of 25 individuals. As we can see from Fig.3, we have been able to reach a maximum score of 259. After that, fitness did not seem to improve. However, the agent has learnt how to walk without falling using both legs. When we have run the best configuration 10 times and averaged it, we have been able to reach a maximum score of 244. In the average best, Fig.3, we can notice a pretty high variance resulting in a bad average performance. So training has appeared to be very unstable. We have also tried to change hyperparameters values in order to achieve a better average performance but it has been proved to be unsuccessful.

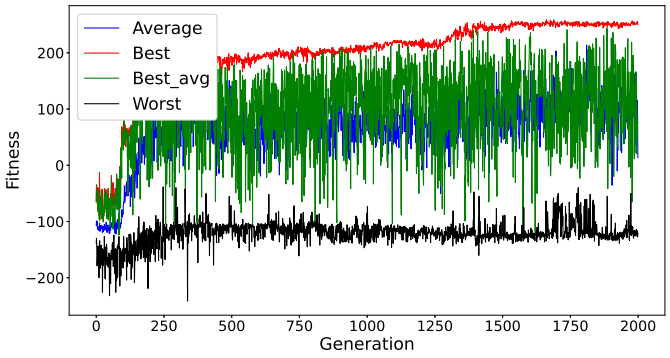

Fig. 3. This is a figure comparing the best, average, worse, and best average fitness scores for each generation for the CMA EA algorithm.

D. TD3

Here both the actor and the critic were implemented using a feed-forward network made of two hidden layers with 512 neurons each. The values of the other parameters, such as the noise and the threshold set to clip the gradients, had the standard values commonly used in literature, as this algorithm is not that much dependent on a particular fine-tuning. Due to the fact that the training was computationally heavy and the rewards did not seem to improve during the last 300 episodes, we have decided to stop the training at episode 800. Regarding the general trend of the graph we can notice that during the first 300 episodes there is not a really significant improvement in the average reward. However, later on it manages to reach a score of 270 in a single episode which is actually the highest score among all the algorithms taken into account. Results show unluckily a high variance, penalizing the performance of TD3. This is a direct consequence of the alternation of low values and periodically peaks during the training phase.

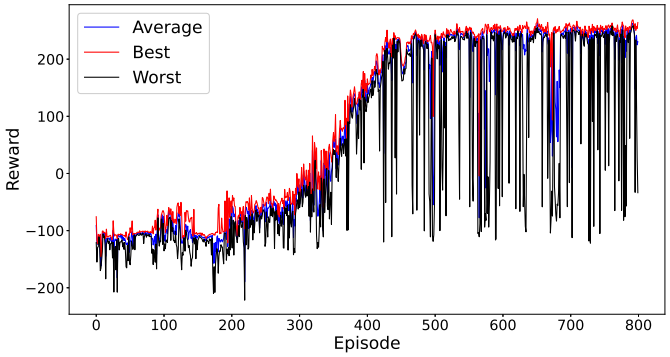

Fig. 4. This is a figure comparing the best, average, and worse reward scores for each episode for the TD3 RL algorithm.

The stagnation of the max score observed at the end of the training may have been caused by a lack of exploration. In fact the agent successfully has learnt how to walk using both legs but probably not in the most efficient way and this results in a fairly high score, but in no further improvements.

IV. Difficulties encountered

One of the biggest problems we had to face was due to the running time of the different algorithms. Google Colab kept logging us out as the maximum space of the RAM was reached, and so it was not possible for example to increase the number of generations as well as the number of steps of Bipedal Walker which would have led to better results. We have tried to tackle these difficulties by running evolutionary codes locally using also parallelization.

V. Lessons learnt

We have proven here that there is not a unique way to solve a certain task. Each one the technique has its pros as well as its cons. Reinforcement Learning is concerned with a specific optimization problem and specifically TD3 is a quite complex algorithm designed mainly for continuous environments. On the other hand GAs and ES are self-learning more general algorithms. Results obtained by running them suggest that sampling in the region around good solutions is often sufficient to find even better solutions. GAs, ES, and NEAT have also improved performances due to temporally extended exploration, meaning they explore consistently since all actions in an episode are a function of the same set of mutated parameters. In this way, an agent takes the same action each time it visits the same state, which makes it easier to learn whether the policy in that state is advantageous.

VI. Conclusions

Summing up everything, what we have tried to do in this project was to make a fair comparison between algorithms. Even if as already stated in the abstract there is not a striking winner, TD3 has proven to be maybe the most suitable algorithm for this dynamical and continuous environment. Evolutionary and genetic strategies would have shown their potential of exploration whether if they would have been tested on the hardcore version of the environment or just if parameters would have been set with a more appropriate fine-tuning. Unluckily, due to the lack of resources we had in our computers, it was not possible.

VII. Contributions

Regarding the way we have divided the work between ourselves, we have proceeded as follows: Matyas has implemented the RL part, Sara the GA, Mostafa the CMA-ES and both Sara and Mostafa have worked on NEAT.

References

[1] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1703.03864

[2] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools Appl.*, vol. 80, no. 5, p. 8091–8126, feb 2021. [Online]. Available: https://doi.org/10.1007/s11042-020-10139-6

[3] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation," in *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996, pp. 312–317.

[4] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[5] S. Zhang and O. R. Zaiane, "Comparing deep reinforcement learning and evolutionary methods in continuous control," 2017. [Online]. Available: https://arxiv.org/abs/1712.00006