Comparison of the behaviour of several algorithms applied to OpenAI GYM environments

Sergi Albiach Caro SERGI.ALBIACH@ESTUDIANTAT.UPC.EDU

Benet Manzanares Salor BENET.MANZANARES@ESTUDIANTAT.UPC.EDU

Joaquim Marset Alsina JOAQUIM.MARSET@ESTUDIANTAT.UPC.EDU

Ramon Mateo Navarro RAMON.MATEO@ESTUDIANTAT.UPC.EDU

Abstract

In this project, we have used Reinforcement Learning (RL), as well as Genetic Algorithms (GA), to try and solve some environments OpenAI Gym offers. Given that the RL algorithms are based in Neural Networks, we have covered quite a few concepts from two parts of this subject.
We implemented and trained a PPO, a DQN and a genetic algorithm to fit the following OpenAI environments: Cart Pole, Breakout and Bipedal Walker. We then compared the results of each of them in terms of metrics like performance (fitness) and time.
Keywords: OpenAIGym, GA, PPO, DQN

1. Problem statement and goals

OpenAI Gym is an open-source Python library that contains different types of environments where RL and GA algorithms can be used to solve them. These environments have different purposes, going from simple Atari games, to more complex tasks like manipulating a robot arm. Because of the broad possibilities it offers, Gym environments have been quite used as benchmark for different algorithms. For this reason, in this project we are trying to apply the concepts we have learnt in this subject, and try to solve some environments with existing algorithms. Therefore, in this project we do not seek to produce some kind of novelty, but rather, apply well-known algorithms in well-known environments.

In particular, we are going to solve the classical **Cart Pole** OpenAI (2021c), the Gym **Bipedal Walker** OpenAI (2021a), and the Atari game **Breakout** OpenAI (2021b).
We are going to solve them using the RL algorithms **Deep Q-Learning** (DQN, Mnih et al. (2013)) Mnih et al. (2013) and **Proximal Policy Optimization** (PPO, O. (2017)), and a **Genetic Algorithm** based on what we have learnt in class.

As mentioned, the goal of our project is not any kind of novelty, but we want to solve the mentioned environments, or at least get the maximum score as possible in each of them,

and perform a comparison of the results.
We have to take into account that inherently some algorithm will be slower than others, or some is well-known to have a big improve in terms of performance than others.

2. Previous work

As mentioned, there is a lot of work done with respect to using the mentioned algorithms to solve the mentioned environments. Not only some algorithms have been bench-marked with the OpenAI Gym environments, but there are a lot of people who have tried to implement these algorithms following the original paper, solving them.
For example, both the DQN and PPO algorithms were tested using Gym environments when they were developed. The results can be found in the respective papers O. (2017) Mnih et al. (2013).
Genetic approaches are less common as they are a kind of brute force technique for this kind of problem, given that the goal is to optimize a neural network. Several individuals have developed their own implementation like Jankowski (2021) among many others but there are few papers regarding this specific approach.

3. The environments

For this practice, we used the mentioned three environments from OpenAI Gym for comparing and contrasting different types of algorithms. We implemented a new class for each environment that extends the base methods to provide a unified interface, calling the original environment functions, returning only the needed information.

3.1. Bipedal Walker

The objective of this environment is to make the robot learn how to walk. A reward is given for moving forward, achieving more than 300 points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small number of points, so a more optimal agent will get a better score.

- **Observation space**: The state consists of hull angle speed, angular velocity, horizontal speed, vertical speed, the position of joints and joints angular speed, legs contact with the ground, and 10 LIDAR rangefinder measurements. There are no coordinates in the state vector.
- **Actions**: Bipedal Walker has 4 joints. the size of our action space is 4 which is the torque applied on 4 joints. The range of value for these actions is [-1, 1]
- **Reward**: The agent gets a positive reward proportional to the distance walked on the terrain. It can get a total of 300+ reward up to the end. If the agent tumbles, it gets a negative reward of -100;

3.2. Breakout

The goal in this game is to maximize the score in the Atari game Breakout. In this environment, the observations are directly the frames from the Atari simulator.

- **Observation space**: The observation space on Breakout is a stack of frames from the game. For simplicity, the image is reshaped in order to improve the computation time.
- **Actions**: For simplicity, the only possible actions that we consider are: move left or move right.
- **Reward**: The game engine offers rewards when bricks are hit. The reward for hitting bricks depends on the position of the brick. The bricks that are higher than others offer a greater reward since they are more difficult to access.

3.3. Cart Pole

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over.

- **Observation space**: Composed by 4 observations that indicate the cart position, cart velocity, pole angle, pole angular velocity.
- **Actions**: The actions are 0 for pushing the cart to the left or 1 for pushing the cart to the right.
- **Reward**: A reward of +1 is provided for every timestep that the pole remains upright including the termination step. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

4. The CI methods

Once we decided that we wanted to combine RL and GA, we had to think about what methods to use. With Genetic Algorithms there was not much to discuss, given that we only known what we have learnt in the subject. However, given that some of the members knew a bit about Reinforcement Learning, we decided to implement some methods which we had already worked before. DQN (Mnih et al. (2013)) is a well-known method that was the state of the art almost 10 years ago. On the other hand, PPO (O. (2017)) is more recent, and outperformed some other methods which in turn outperformed DQN.
For this reason, we decided to implement these 2 different RL algorithms, and try to see how well they perform in the selected environments.

4.1. Genetic

Genetic Algorithms (GA) are a type of Evolutionary Algorithms. These algorithms are based on Evolutionary Computation, a technique whose main characteristic is simulating the computation process as the natural evolution that happens in nature. The evolution cycle that happens naturally goes as figure 1 describes. A subset of individuals of the original population are selected as parents of the next generation, then they are recombined between them and some descendants may be mutated. These recombined and mutated individuals are the offspring. Then this offspring replaces partially or totally the original population.

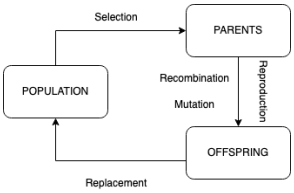


Figure 1: Evolutionary cycle in nature

In GA, the individuals are string of length n, selection is proportional to fitness, the mutation is like a transcription error and the recombination is called crossover. In our particular implementation, each individual is representing a neural network and the crossover and mutation steps are performed over the weights and biases of the network, implemented as torch networks.

Selection could have been done at random, selecting the best individuals or selecting by the Roulette Wheel approach. This approach assigns each individual a probability to be selected as parent proportional to its fitness. So the better the fitness, the highest the probability it is chosen to reproduce. We decided to keep this approach as it balances better the exploration and exploitation.

Crossover can also be done in several ways, a single point crossover, where a random point of the chromosome is selected and the parents exchange one of the parts. We could also consider a two-point crossover or a uniform crossover, where basically each bit of the chromosome is exchanged with a probability of 0.5. We finally decided to keep the single point mutations as it is simpler and has offered good results.

Offspring could replace totally the previous generation or we could establish some criterion to decide which individual to replace, that could be the oldest individuals, the worst ones or even at random. In our implementation, for each new individual in the offspring, we replaced one individual of the older generation if its fitness was lower.

The algorithm stops when the individuals reach a maximum number of generations. We tuned the maximum number of generations according to each problem.

Basing our solution on Robert Jankowski's code (Jankowski (2021)), we implemented our own version with tuned parameters and extra layers of complexity where it was required. It is important to remark that the solution could have been generalized and we could have created an "Individual" class where all the current functionality is contained. Nevertheless, for our convenience and due to time constraints, we decided to keep two separated implementations even though they are very similar.

As has been said, each individual represents a neural network, in particular a perceptron with a single hidden layer. Each environment requires its own input and hidden layers sizes, as well as the output one, to be adapted to the possible actions. Also, as has been stated before, the selection is made through the roulette wheel approach, the crossover is done through a single point and replacement is done by removing individuals that perform worse than the offspring.

4.2. DQN

Deep Q-Network (DQN, Mnih et al. (2013)) is a Deep Reinforced Learning approach based on the popular Q-learning algorithm (Watkins and Dayan (1992)). In particular, it uses neural networks to learn the Q-function, which for a state-action pair returns the expected reward. This function is later used for selecting the action with a greater expected reward for the current state since it is considered the best action.

In classical Q-learning, this Q-function is learned using the Q-table which contains the Q-value for each state (as rows as possible states) and action (one column for each possible action). This table is initialized with zeros and it is updated throughout interaction with the environment and the Bellman equation:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 2: Bellman equation for Q-values update

Note that the discount factor describes how much we thrust the new value, fundamental for stability since the next values can significantly change as the policy improves.

This training process is capable to learn a specific policy. Nevertheless, this policy can be far from optimal and get locked up there doing always the same actions. To attach this problem, an Epsilon-Greedy Exploration method is defined, performing random actions with an epsilon probability (which decreases along with the training), discovering new situations and exiting the suboptimal loop.

The main problem of Q-learning is that it requires a Q-table with as many rows as possible states, rapidly scaling to unaffordable memory requirements. On this basis, it is not applicable to complex environments with big state spaces.

In the DQN method, the Q-function is modelled using a neural network, which receives the current state as input and outputs the expected Q-value for each possible action. In this way, the goal of the network is both to compute an embedding for the input state and the Q-value for each one. Consequently, solving the problem of Q-learning regarding the size of the Q-table. In addition, it also uses the Epsilon-Greedy Exploration method.

During training, this approach defines two Q-networks, the policy (or main) and the target, which are initialized identically. The policy network is used for interacting with the environment, selecting the action with a greater Q-value at each step (except if a random exploratory action is ordered). On the other hand, the target network is exclusively employed for training, being its predictions the old Q-values to use at the Bellman equation. Concretely, each optimization step consists of processing state-action pairs with already-known rewards with both networks, computing the Bellman equation using rewards and target network's outputs (as old Q-values) and using the results as the expected output for the policy network (with the loss function). Back-propagation and gradient descent optimization is only performed for the policy network, learning the Q-values. The target

network is updated as a copy of the policy once every a certain amount of episodes or steps (more infrequent than policy network update). This approach may seem confusing or peculiar, but the authors mention that this two-networks approach makes the learning more stable and consistent.

Another characteristic of this method is how training examples are gathered. To avoid using consecutive states as inputs (hard to differentiate), the authors define a Replay Memory, which stores the last M states (being M significantly greater than the batch size) and allows creating mini-batches from random previous states. Additionally, each sample is formed by a reduced amount of consecutive states (usually four), allowing the model to detect dynamics such as movement.

In Mnih et al. (2013) and Mnih et al. (2015), the DQN algorithm was evaluated on multiple Atari games (including Breakout) obtaining outstanding or even above human results. It is important to note that this was achieved only by receiving as input the game's frames and rewards, identical to how it would be for a human. This is done using a CNN with two or three convolutional layers, a flatten operation and two fully connected layers; providing the latter the output regarding each action.

In summary, DQN was an important step forward for Reinforcement Learning in 2013, showing the possibilities of neural networks in the field. However, currently, multiple approaches improve its results, such as DoubleDQN Van Hasselt et al. (2016) and PPO O. (2017).

4.3. PPO

Proximal Policy Optimization (PPO) is a Reinforcement Learning algorithm belonging to the family of Policy Gradient algorithms, in particular, to the actor-critic ones.
Actor-critic algorithms are those that directly learns a policy (i.e. a probability distribution for each state) instead of extracting it from the state-value function ($V(s)$), or from the state-action value function ($Q(s, a)$). The idea is using the critic (state-value function) to guide the learning course of the actor (policy).

To guide the learning, typically the advantage function is used (i.e. $A(s, a) = Q(s, a) - V(s)$), which is plugged into the actor's loss. If for a particular state, taking a particular action was not good, which will result in a low advantage, the probability for that action will be reduced. Instead of using the advantage, we have implemented the Generalized Advantage Estimation P. (2018), which usually performs much better.

With respect to the loss functions PPO uses, for the critic is pretty much the Mean Squared Error (MSE), which we will try to minimize. The actor loss, which we want to maximize, is slightly more difficult. The basic loss function of Policy Gradient is defined as $E_t[log\pi_\theta(a_t|s_t)A_t]$, being π_θ the policy function defined with the parameter θ, and A the advantage function. This loss has the problem of possibly causing too large policy updates, causing the performance to collapse. PPO tries to solve this problem by constraining the update. The way PPO does it is by defining the Clipped Surrogate Objective:

$$L_{CLIP}(\theta) = \mathbb{E}(min(r_t(\theta)A_t,\ clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$$
$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

In this equation, r_t is a ratio of an action's probability, representing how much the policy has changed that probability between update steps. What PPO does is ensure the policy does not change too much by clipping the ratio between $1-\epsilon$ and $1+\epsilon$ (being ϵ a hyperparameter).
However, even though PPO can constraint the policy update, is not completely reliable, and sometimes can fail. That is why some improvements can be done using the Kullback-Leibler divergence. A simple one consists in not updating the policy if the value is higher than a certain threshold (e.g. 0.03). In X. (2019), some other modifications to PPO based on the KL divergence are explained.

Another characteristic this algorithm has is the fact that it uses the same batch of samples to train the networks in multiple epochs. Thanks to constraining the size of the policy update step, this algorithm is more data-efficient, and can perform multiple updates with the same samples.

The last characteristic to take into account is the use of multiple instances of the environment running in parallel during the training process. The agent interacts in all the environments at the same time during several steps, storing the sampled trajectories in a buffer. Then, the stored transitions are merged and shuffled, and the training epochs are done for both actor and critic. Once the iteration is finished, another is started, the buffer is cleared, and the process of gathering data starts again. This way of proceeding is called On-Policy. This way we ensure we are not updating with trajectories sampled with an old policy.

5. Results and Discussion

In this section, we present the results obtained for each environment, applying each algorithm. A direct comparison is not trivial, since methods' experience with the environments are defined by parallelization methods (multiple agents interacting with multiple environments), training methods differ and not all environments are used for everyone. Consequently, contrasting will be made taking into account the converged models (if possible) in the Cart Pole environments (the only common). In particular, we are going to compare in terms of the evolution of the episodic reward during the course of training, the number of episodes needed to reach those results and the stability of the learning process.

5.1. GENETIC

In order to test the genetic approach, we have used the CartPole and Bipedal Walker environments. These two environments have a similar observation space, as it is a set of observations. Breakout was discarded because the observation space was a stack of images

and therefore a more complex approach like a convolutional neural network must have been used and thus the crossover and mutation steps are more complex in these networks and even though we tried to implement it, we were not successful.

For every plot we can see in green the maximum, in orange the mean and in blue the minimum fitness of the population up to every generation.

5.1.1. Cart Pole

Cart Pole has been tested with different combinations of parameters. For the generations, we tested, 5 10 and 25. For the population, we used 10 30 and 50. For the mutation rate we tried out 0.2, 0.4, 0.6 and 0.8. Finally, for the crossover rate, we tested 0.2, 0.5, 0.8 and 0.9. We cannot show every combination, so we chose two that we considered worked best, figures 3 and 4.

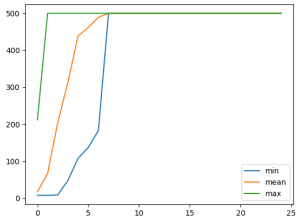


Figure 3: 25 generations - 50 population - 0.6 mutation rate - 0.8 crossover rate

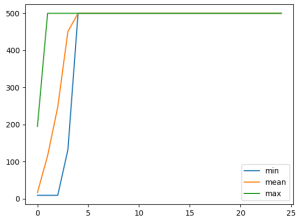


Figure 4: 25 generations - 30 population - 0.2 mutation rate - 0.8 crossover rate

From the several runs, we can say that the higher the population, the faster a good solution is found. Crossover is very important, as the best runs always have 0.8 of crossover rate. For this particular problem, the mutation rate didn't seem to affect very much, probably because the solution is found quickly and there is no need for exploration.

The maximum fitness (500) is achieved in less than 5 generations, and the rest of the individuals soon converge in less than 10 generations. We can see that once the best fitness is found, the individual with minimum fitness is replaced by the offspring and little by little starts to grow. This is because the replacement method we implemented by which individuals with lower fitness than the offspring are replaced.

5.1.2. Bipedal Walker

Bipedal Walker has also been tested with different combinations of parameters. Initially we tested for the generations, 100 and 200. For the population, we used 50 and 100. For the mutation rate we tried out 0.4, 0.6 and 0.8. Finally, for the crossover rate, we tested 0.4, 0.6 and 0.8.
As the results were not very promising, we kept the best combination of parameters: 100 for the population size, 0.8 for crossover rate and 0.6 for mutation rate, but we increased

the maximum number of generations to 4000. The results obtained are shown in figure 5.

As a note, the 0.6 mutation and crossover combination plots are not included because the code was re-executed without that option.

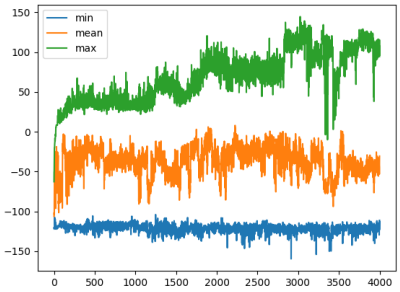


Figure 5: BipedalWalker run with: 4000 generations - 100 population - 0.6 mutation rate - 0.8 crossover rate

For this problem, the maximum fitness achieved is around 140 and 3000 generations are needed to obtain this value. It is also seen from this plot that this game present a lot of instability because even tough we are replacing the worst individuals, there is always at least one individual that falls and get a reward of -100. Nevertheless, the best individual is seen to get better and better over the generations, with counted exceptions where it fails and suddenly drops the performance.

5.2. DQN

For experiments with DQN, only the Breakout and Cart Pole environments are used, employing their rendered frames as states representations instead of the predefined ones. Bipedal Walker was discarded as it raises an unsolved exception during rendering, not being possible to obtain the current frame.

Regarding the implementation, the code from et al. (2021) is used as base, adapting it to be as faithful as possible to the defined at the original paper Mnih et al. (2013). In addition, we aim to use the hyperparameters defined at Mnih et al. (2015), but we note two important issues. First, the DQN defined at Mnih et al. (2015) is bigger than that from Mnih et al. (2013), with one more convolutional layer and more neurons at the first fully connected layer. The second, is that the process used for training required from parallelization (multiple agents with multiple environments at the same time) and that was not possible due to hardware restrictions (both memory and processing capacity). It is important to note that the DQN approach requires from two CNNs and the replay memory space, so it is computational heavier than the other methods.

In order to approximate the real context, we defined a "Base" and "Big" models based on the specifications from Mnih et al. (2013) and Mnih et al. (2015) and an independent hyperparameters search (considering, learning rate, momentum, steps between updates and

epsilon-greedy policy) was carried out for both models and both environments. In particular, evaluations with 200 and 1000 episodes were made. Finally, we conclude that, as far as we were able to train, both models performed similarly and different hyperparameters are required for both environments.

As a consequence, the following evaluations are made with the "Base" model (smaller, so faster to train) using the best found hyperparameters for each environment (which can be found at the corresponding results' folder). Model convergence is not always achieved due to hardware limitations, making training as long as possible within reasonable time periods (multiple hours).

5.2.1. Cart Pole

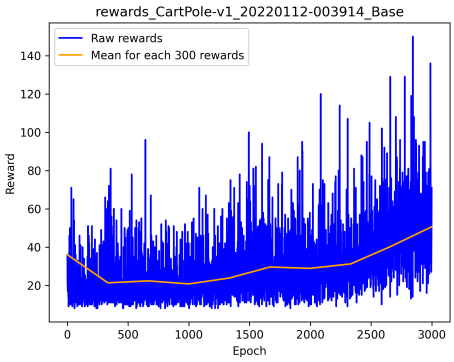


Figure 6: Cart Pole results using DQN for 3000 episodes

As can be noticed in Figure 6, the model is learning slowly but constant, reaching a maximum average reward of 45. Nevertheless, these results are significantly worse than those obtained with others approaches, which reached the maximum of 500 points. One reason for this fact is the difference in how environment is perceived. While other methods receive a vector with specific information about the cart and pole status as input, our DQN has to infer these from the images; adding a not negligible difficulty. The other related and more important cause is the lack of training. It can be observed that the model keeps improving its results at the last episode, meaning that more training probably will keep improving the average reward up to at least competitive values. With more resources, it would have been possible to parallelize the training to perform more episodes in a reasonable amount of time, but these were not available.

5.2.2. Breakout

Figure 7 depicts a slow but noticeable learning process, reaching an average of 2 points after 3000 episodes. This result is notoriously lower than the 401.2 ($\pm$26,9) obtained at Mnih et al. (2015) (1327% the 31.8 obtained by humans). We suppose that this is due to the usage of the smaller "Base" model instead of the "Big" used at the article, the different hyperparameters, and the lack of training. In particular, the authors at Mnih et al. (2015) trained their models for 10 million frames, what would be equivalent to 40000 episodes if

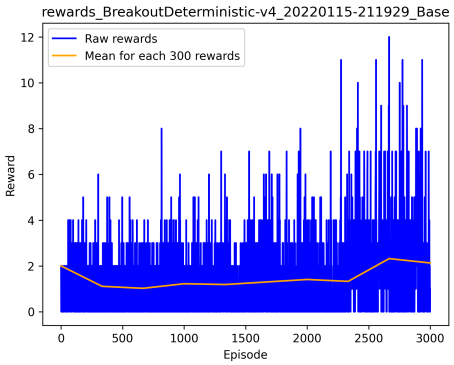


Figure 7: Breakout results using DQN for 3000 episodes

keeping the average of 250 steps (as was the average for our results). Again, with more computation resources these results could have been better.

5.3. PPO

The code of the PPO algorithm has been developed taking as reference the code from the Spinning Up resources from the people of OpenAI OpenAI (2018), simply changing the way we manage the multiple environments, the way we constraint the policy change with the KL divergence, and we have also added a learning rate decay.

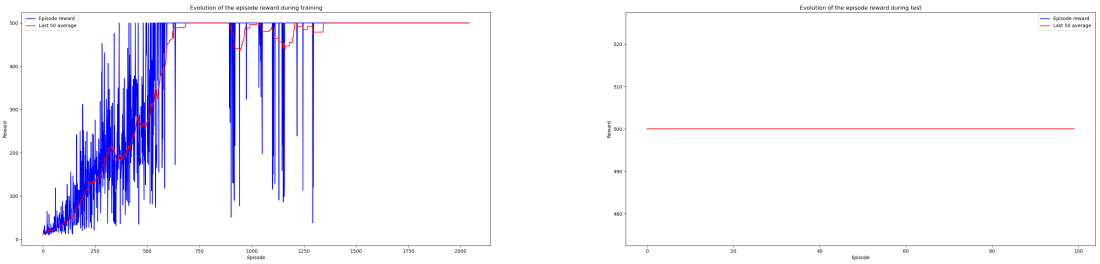
Concerning the hyperparameters, we have mainly changed the number of environments, the number of steps per iteration, the buffer size, the number of PPO epochs, and the learning rate. The other parameters have been remained unchanged between the executions. The best found for each environment are commented in the code.

Finally, we also want to mention that PPO is not trained in terms of episodes, but rather iterations. Given that we are using multiple environments at the same time, and each ends an episode at different moments, we cannot formulate it as performing a certain number of episodes. That is why in each environment, a different number of episodes has been performed. Also, we have to say that the training process has been stopped when the average reward of the last 50 episodes does not improve anymore.

5.3.1. Cart Pole

Cart Pole has been trained during 2000 episodes, given that it was able to converge, and later tested with another 100 episodes using the best-achieved weights.

As it can be observed, with 700 episodes the agent was able to completely balance the pole, reaching that point with quite instability, achieving the maximum score of 500 continuously during some episodes. The problem lies in the fact that the policy started to diverge, causing the pole to fall during some episodes. From that point onward, it took almost 500 episodes to converge again and stabilize until no more changes happened.

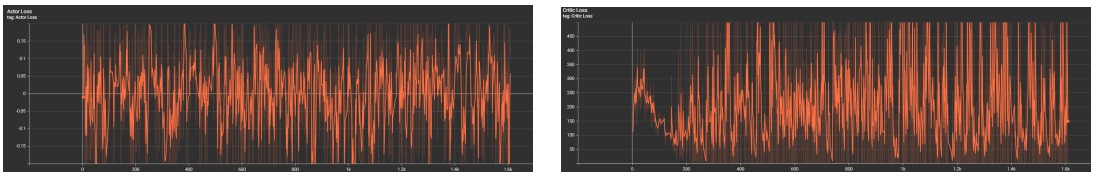


(*a*) Train curve (*b*) Test curve

Figure 8: Cart Pole train and test curves

Then, at test time, we can see how it perfectly balances the pole during all the episodes, reaching always the maximum score.

Trying to determine which could be the cause to that divergence is difficult, because both losses do not seem to dramatically change during the training phase, specially at that point where started to diverge. That is why we are going to consider that is mainly because at that point it did not perfectly mastered the task.



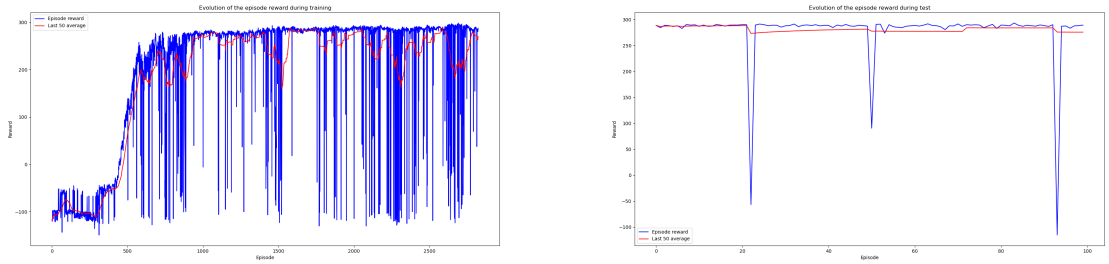
(*a*) Actor loss (*b*) Critic loss

Figure 9: Cart Pole Actor and Critic loss

5.3.2. Bipedal Walker

Bipedal Walker has been trained during almost 3000 episodes, and again tested for 100 more episodes with the best-achieved weights.

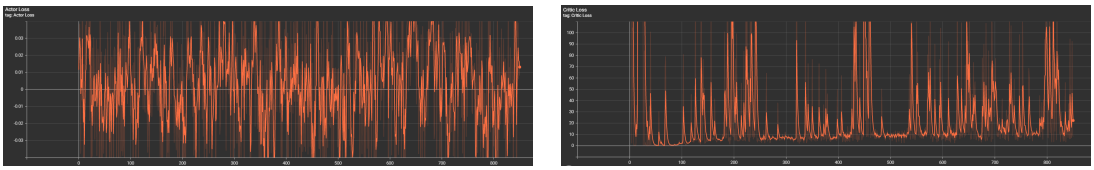
Observing the red curve, the agent can start learning how to walk in 500 episodes, constantly improving until reaching 200 points. There, the learning pace is slowed down, being more difficult to improve. The red curve is more or less stable between the 200 and 300 points, never being able to converge to the maximum 300 points where the game is cleared. If we observe the blue curve, we can see how the agent is really unstable. Most of the time the agent indeed achieves around 280 points, but a lot of times the agent falls, automatically getting -100 points, and causing those drops. Therefore, the agent has more or less learnt how to walk, but it is not robust at all to avoid falling.
During testing, we can see how is performing good most of the time, but it ends up falling sometimes, based on the randomness of the course. We have to mention that the best weights are achieved around the 1500 episodes, so from that moment onward, the training phase has not been able to improve nor stabilize during the remaining episodes.



(*a*) Train curve (*b*) Test curve

Figure 10: Bipedal Walker train and test curves

This instability we see is attributed to the fact that the critic loss suddenly increases, causing the policy to dramatically change. Even though we are constraining with the KL divergence, is not enough.



(*a*) Actor loss (*b*) Critic loss

Figure 11: Bipedal Walker Actor and Critic loss

As it can be observed, the actor loss does not present that much of a change, to be expected. However, the critic loss is sometimes suddenly increasing when it was already quite low. This corresponds to those episodes where the agent falls, and it started performing worse during some episodes until restoring.

6. Strengths and weaknesses

Regarding the genetic approach, we can state that in some cases, like the Bipedal Walker environment, takes a lot of time to find an acceptable solution due to the nature of the approach (more than 40 hours for the 4000 generations). Even tough the provided solution is quite good, it is true that it should have been generalized to accept other kind of networks easily. Moreover, more parameters should have been added to test them out and compare them more thoroughly.

For the DQN approach, we note that having more computational resources and a parallelization methodology would have allowed us to extend the training further and thus explore the true capabilities of the method in these environments. An alternative would be to use a simpler state representation than frames (reducing cost), but it would no longer be in the same conditions as a human.

PPO is the one requiring more resources given that it needs a thread for each parallel environment and also stores more data in the buffer. However, it is able to achieve relatively good performance in a small amount of time, given that the training process did not last more than 2 hours for the tested environments. The problem is the learning process being quite unstable, not being able to converge in the Bipedal Walker.

7. Conclusions and future work

The first thing to conclude is that using OpenAI Gym environment is not as easy as one may think at first. For some games like Breakout, a lot of installations are needed and it is not a straightforward process in general. In addition, multiple GitHub links from the environment documentation are broken.

Regarding the environments, it has been clear to us that the easiest game to learn is Cart Pole, probably because there are few actions to perform, the concept is more or less simple, and we can reach the maximum fitness quite fast. Bipedal Walker was moderately easy to learn too, but can be more complicated than Cart Pole. Finally, Breakout is the most difficult one, as the stack of images should be interpreted to find out the direction of the ball.

Regarding genetic approaches, it has been clear to us that they can be very useful and even efficient in some cases, but for other more complicated problems the time they require to perform all the generations is prohibitive, even more when there are other approaches that reach the same performance in far less time. Moreover, complex networks can be hard to deal with and deep knowledge of its working is required to correctly implement the algorithm. As future implementations for this part of the work, a testing method could be implemented to test the best network and obtain visual feedback of the trained model.

For the DQN method, it has been surprising capable of learning from the environment frames, even though its results are not as good as expected. The hyperparameters search has proved to be a fundamental part of the methodology, changing from a bad unstable model with catastrophic forgetting to a learning-capable model. In future work, it is needed to use more computational resources, parallelization and time to solve the lack of training and explore the true capabilities of the approach in the environments.

Regarding PPO, it has been able to learn to perform well in the environments in a short amount of time, achieving the best overall results. However, it has been shown that some instability exists. Finding the correct network architecture, hyperparameters, and constraining the policy update to avoid the network crashing, have been the most important parts. As future work, we should need to try more approaches to constraint the critic loss, which in turn affects the actor loss, and end up messing with the current policy, causing it to diverge. We could try not to perform updates to the network if the loss is high, use some scaling in the reward. We could also try modifications of PPO, like the ones mentioned in X. (2019) could be tested. Other approaches could be to try to sample more data with more parallel environments, implement a more intelligent learning rate decay, or try a different network architecture.

References

Brian Johnson et al. Pytorch dqn tutorial. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html, 2021.

Robert Jankowski. OpenAI GYM using genetic algorithms. https://github.com/robertjankowski/ga-openai-gym, 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, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Schulman J. Wolski F. Dhariwal P. Radford A. Klimov O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

OpenAI. Proximal policy optimization. https://spinningup.openai.com/en/latest/algorithms/ppo.html, 2018.

OpenAI. OpenAI GYM bipedal walker environment. https://github.com/openai/gym/wiki/BipedalWalker-v2, 2021a.

OpenAI. OpenAI GYM breakout environment. https://gym.openai.com/envs/Breakout-ram-v0/, 2021b.

OpenAI. OpenAI GYM cartpole environment. https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py, 2021c.

Schulman J. Mortiz P. Levine S. Jordan M.I. Abeel P. High-dimensional continuous control using generalized advantage estimation. *ICLR*, abs/1506.02438, 2018. URL https://arxiv.org/pdf/1506.02438.pdf.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

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

Wang Y. He H. Tan X. Truly proximal policy optimization. *CoRR*, abs/1903.07940, 2019. URL http://arxiv.org/abs/1903.07940.