On the comparison between Q-Learning and Policy Gradient on the Cart-Pole problem

IEOR 265, University of California, Berkeley

Final Report

Jules Bertrand[1], Adam Moutonnet[2]
[1]jules.bertrand@berkeley.edu, [2]adam_moutonnet@berkeley.edu

I. Introduction

In an RL problem, intelligent agents evolve following a discretized horizon in an environment in which they take intelligent actions at each step according to their observations. Each of these actions has an influence on the environment, changing its state step by step, and consequently the observations that agents make of it. We can assign an objective to each agent, which he will try to achieve through a sequence of actions that he will undertake. In order for the agent to learn which action to take based on his observation of the environment, we introduce the concept of rewards and penalties. Depending on how the action he or she takes changes the state of the environment, he or she is given a reward or a penalty. The reward means he is on the right track, the penalty means he is off track. Therefore, the objective of the agent at each step is to take the action that maximizes the expected future reward.

The most common analogy of reinforcement learning problems is games. In games, the agents are the players and they try to take the best action at each step to maximize their odds of winning the game. If they lose a game, they will understand that the actions they took were not so good (it is a penalty). On the other hand, if they win it means that their actions were not so bad (it is a reward).

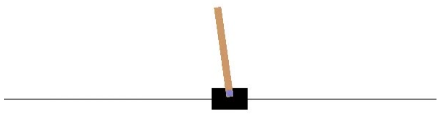
Fig. 1: Screenshot of the game CartPole-V0 from *Open AI*

That is why the Cart-Pole problem (represented on Figure 1) is widely used to demonstrate the performances of reinforcement learning algorithms. The principle is really easy. A pole attached to a cart is balanced over this cart, and the purpose is to push the cart left or right so that the pole remains balanced for the greatest number of steps. In addition, the impressive Python library *Open AI* implements this environment, making it accessible to everyone.

Our goal throughout this project is to explore the two main reinforcement learning methods in use nowadays, which are *Q-Learning* [17] and *Policy Gradient* [15]. We will first cover all the mathematics behind the two methods, starting with the Vanilla version and ending with modified, much more advanced versions of these two. We will then describe how to implement both methods by describing the precise algorithm behind them. Finally, after having trained our different neural networks, we will compare their performances on some criteria and highlight the advantages and drawbacks of each method.

II. Description of the Methods

Let $N \in \mathbb{N}^* \cup \{+\infty\}$ be our discretized horizon, and $k \in [|0, N|]$ be the current step. In our problem, we will have only a single agent evolving in the environment. Let $\mathcal{U}$ be the space of actions the agent can take in the environment, and $\mathcal{X}$ be the space of states the environment can be in. Both $\mathcal{U}$ and $\mathcal{X}$ are discrete and bounded in the case of Cart-Pole or most Atari games. At step k, let $x_k \in \mathcal{X}$ be the state of the environment, and $u_k \in \mathcal{U}$ be the action taken by the agent. We consider that the agent always has access to the full state of the environment. We define:

$$\begin{aligned} f: \quad & \mathcal{X} \times \mathcal{U} \to \mathcal{X} \\ & (x_k, u_k) \to x_{k+1} \end{aligned}$$

the function which determines next state x_{k+1} based on the current one x_k and the action taken u_k. This function is fully deterministic but a priory not known (we do not know the dynamics of the game). In the case of Atari games, the initial state of the game x_0 is fixed and always the same. We also define the reward function:

$$\begin{aligned} r: \quad & \mathcal{X} \times \mathcal{U} \to \mathbb{R} \\ & x_k, u_k \to r_k \end{aligned}$$

which determines the reward r_k earned at step k by taking the action u_k in state x_k. We refer to as a policy π any probability distribution of actions conditioned on a state such that $\forall k \in [|0, N|], u_k \sim \pi(\cdot \mid x_k)$. We call a trajectory any sequence of state-action

$(x_0, u_0, x_1, u_1, \dots)$, the length of which we don't know a priori, and we denote by τ such trajectory. This trajectory has some probability to happen under a given policy π, which we denote by $\pi(\tau)$. Let $\gamma \in [0,1]$ be the discounting factor, we define the discounted reward $R(\tau)$ associated to a trajectory τ:

$$\begin{aligned} R: \quad & \mathfrak{T} \to \mathbb{R} \\ & \tau \to \sum_k \gamma^k r(x_k, u_k) \end{aligned}$$

Where $\mathfrak{T}$ is the space of all possible trajectories. We also define the Q-value associated to a state-action pair and a policy: this is the discounted future reward we can expect by taking action u_k in state x_k following a policy π:

$$\begin{aligned} Q^\pi: \quad & \mathcal{X} \times \mathcal{U} \to \mathbb{R} \\ & x_k, u_k \to \mathbb{E}_{\substack{x_{k+1:} \\ u_{k+1:}} \sim \pi} \left[\sum_{j \geq k} \gamma^{j-k} r(x_j, u_j) \mid \pi, x_k, u_k \right] \end{aligned} \tag{1}$$

Q-values can be computed recursively with dynamic programming:

$$\forall (x_k, u_k) \ \mathcal{X} \times \mathcal{U} \quad Q^\pi(x_k, u_k) = r(x_k, u_k) + \gamma \mathbb{E}_{\substack{x_{k+1} \\ u_{k+1}} \sim \pi} \left[Q^\pi(x_{k+1}, u_{k+1}) \mid \pi, x_k, u_k \right]$$

With $x_{k+1:} = x_{k+1}, x_{k+2}, \dots$ and $u_{k+1:} = u_{k+1}, u_{k+2}, \dots$. And finally we define the Value function, which is the discounted future reward we can expect by being in state x_k if we follow a policy π:

$$\begin{aligned} V^\pi: \quad & \mathcal{X} \to \mathbb{R} \\ & x_k \to \mathbb{E}_{\substack{x_{k+1:} \\ u_{k:}} \sim \pi} \left[\sum_{j \geq k} \gamma^{j-k} r(x_j, u_j) \mid \pi, x_k \right] \end{aligned} \tag{2}$$

The advantage function describes how much better or worse an action is on average than others when the agent is following the policy π. We define it as:

$$\begin{aligned} A^\pi: \quad & \mathcal{X} \times \mathcal{U} \to \mathbb{R} \\ & (x_k, u_k) \to Q^\pi(x_k, u_k) - V^\pi(x_k) \end{aligned} \tag{3}$$

A. *Policy Gradient*

REINFORCE

Let θ be some parameter (of a neural network or any other similar mathematical tools). We define π_θ which is the probability density function of a given policy parametrized by θ. $\forall k \in [|0, N|]$,

$$u_k \sim \pi_\theta(\cdot \mid x_k)$$

In other words, the neural network with parameter θ takes as input the current state of the environment x_k, and outputs $\pi_\theta(\cdot \mid x_k)$ the probability distribution followed by u_k. For example, in an environment where the two available actions are left or right (e.g. *Tetris* game), we can imagine that u_k follows a Bernoulli distribution $B(p)$, and that the neural network predicts the parameter p based on x_k. For ease of notations, we assume in the following demonstrations that $\gamma = 1$, but they are also valid for any $\gamma \in [0,1]$ ($\gamma \in [0,1[$ when trajectories have infinite length).

The purpose of Policy Gradient method is to find the optimal $\pi_\theta^* = \pi_{\theta^*}$ that maximizes the discounted reward expectation for a trajectory generated following this policy. Concretely,

$$\theta^* = \arg\max_\theta J(\theta) \text{ with } J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)]$$

By definition of the expectation, we can rewrite $J(\theta)$:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)] = \sum_{\tau \in \mathfrak{T}} \pi_\theta(\tau) R(\tau)$$

Note that here we assume that the probability distribution given by π_θ is discrete, but it works the same way with a continuous distribution using an integral and a density function. Since this is an optimization problem, we need to derive the gradient of $J(.)$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(\tau) R(\tau)] \qquad \textit{Proof in Appendix} \tag{4}$$

Let $P(x_0)$ be the probability to start at the state x_0 and $P(x_{k+1} \mid x_k, u_k)$ be the probability to get in the state x_{k+1} knowing that we started at state x_k and took action u_k. We can derive $\pi_\theta(\tau)$:

$$\pi_\theta(\tau) = P(x_0) \prod_k \pi_\theta(u_k \mid x_k) P(x_{k+1} \mid x_k, u_k)$$

$$\nabla_\theta \log \pi_\theta(\tau) = \sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) \qquad \textit{Proof in Appendix} \quad (5)$$

This very interesting result tells us that the gradient does not depend on the dynamics of the system. That is why we say that Policy Gradient is a model-free method, we do not need to know the dynamics of the system to do the optimization. Thus,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\left(\sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) \right) \left(\sum_k r(x_k, u_k) \right) \right]$$

Now, let b be some value which at some step k does not depends on $u_k, x_{k+1}, u_{k+1}, \ldots$. We want to prove that this has no influence on the gradient. Let $k \in \mathbb{N}$,

$$\mathbb{E}_{\tau \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(u_k \mid x_k) b \mid \pi_\theta] = 0 \qquad \textit{Proof in Appendix} \quad (6)$$

With $x_{0:k} = x_0, \ldots, x_k$ and $u_{0:k-1} = u_0, \ldots, u_{k-1}$. Hence, the gradient can be rewritten by subtracting $\sum_{j<k} r(x_j, u_j)$ whose gradient is zero according to Equation 6:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k) \right] \qquad \textit{Proof in Appendix} \quad (7)$$

Thus, we can derive a surrogate loss $J(\theta)$ by ”integrating” this gradient formula:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k) \right]$$

In the cases where we do not have access to the total number of trajectories (which can be infinite), but only to a subset $\{\tau^{(1)}, \ldots, \tau^{(m)}\}$ of $\mathfrak{T}$, we can approximate $\nabla_\theta J(\theta)$ with a Monte-Carlo technique:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \sum_k \left(\nabla_\theta \log \pi_\theta(u_k^{(i)} \mid x_k^{(i)}) Q^{\pi_\theta}(x_k^{(i)}, u_k^{(i)}) \right)$$

With $\tau^{(i)} = (x_0^{(i)}, u_0^{(i)}, x_1^{(i)}, u_1^{(i)}, \ldots)$. Here, $\log \pi_\theta(u_k^{(i)} \mid x_k^{(i)})$ is the log-likelihood of observing $u_k^{(i)}$ given $x_k^{(i)}$ and some parameter θ. Now that we have the gradient, we are able to carry out a gradient ascent. Let $\alpha \in \mathbb{R}^{+*}$ be some learning rate, the gradient ascent at each step of the optimization process is:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

This method is called *REINFORCE* and is actually not widely used because of convergence issues we will analyze later. To summarize, Policy Gradient tries to find the optimal policy that maximizes the expectation of the discounted future reward, and Equation 7 tells us that it is equivalent to changing the policy so as to increase the likelihood of ”good” actions (actions that has a high Q-value for the current state).

Advantages Actor Critic

One of the best-known challenge with Policy Gradient is the large variance in the processed gradient. This can lead to brutal change in policy during the training, causing convergence issues. One solution to this problem developed in [11] is to add a baseline which will be the value of the state $V^{\pi_\theta}(x_k)$. Indeed this function does not depend on $u_k, x_{k+1}, u_{k+1}, \ldots$ thus will have no effect on the parameter update. However, this function is unknown a priori, therefore we will try to approximate it with a neural network of parameter ψ. We denote by $V_\psi^{\pi_\theta}(x_k)$ this approximated function of parameter ψ. From now on, we consider $\gamma \in [0, 1[$. We define the advantage function relying on ψ:

$$\begin{aligned} A_\psi^{\pi_\theta} : \quad & \mathcal{X} \times \mathcal{U} \to \mathbb{R} \\ & (x_k, u_k) \to Q^{\pi_\theta}(x_k, u_k) - V_\psi^{\pi_\theta}(x_k) \end{aligned} \qquad (8)$$

Which describes how much better or worse an action is compare to other actions on average relative to the policy π_θ. Thus, the objective function becomes:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \log \pi_\theta(u_k \mid x_k) A_\psi^{\pi_\theta}(x_k, u_k) \right]$$

This method is called *Advantages Actor Critic* (or A2C) and is much more used than REINFORCE thanks to its better convergence. The *Critic* is used to approximate the value function $V_\psi^{\pi_\theta}$, and the *Actor* updates the policy parameters in the direction suggested by the Critic. Concerning ψ, it has to constantly converge to its optimum value ψ^* defined by:

$$\psi^* = \arg\min_\psi U_\theta(\psi) \text{ with } U_\theta(\psi) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_k \frac{1}{2}\left(V_\psi^{\pi_\theta}(x_k) - \sum_{j\geq k}\gamma^{j-k} r(x_k, u_k)\right)^2\right]$$

As soon as θ changes, ψ^* is not be the optimal value anymore and the optimization needs to be done again. The gradient can be easily derived as it is a simple mean squared error problem. Concretely, as before, we need to generate a number of trajectories $\{\tau^{(1)}, \ldots, \tau^{(m)}\}$, and $U_\theta(\psi)$ will be approximated:

$$U_\theta(\psi) \approx \frac{1}{m}\sum_{i=1}^m \sum_k \frac{1}{2}\left(V_\psi^{\pi_\theta}(x_k^{(i)}) - \sum_{j\geq k}\gamma^{j-k} r(x_k^{(i)}, u_k^{(i)})\right)^2$$

A more general version of the advantage was developed and is called *General Advantage Estimate*, and also other methods like *Proximal Policy Optimization* or *Entropy Regularization* leading to a much better training efficiency. The mathematical development of these methods can be found in Appendix.

B. Deep Q-Learning

With Q-learning, we aim at predicting the optimal value Q-functions Q^* associated to every possible state-action pairs (x, u), solutions of this equation [1]:

$$\forall\,(x,u) \in \mathcal{X}\times\mathcal{U} \quad Q^*(x,u) = r(x,u) + \gamma\mathbb{E}_{x'\in\mathcal{X}}\left[\max_{u'\in\mathcal{U}} Q^*(x',u')\big| x, u\right]$$

Hence, the best available action u^* to take when you are at state x will simply be:

$$u^* = \arg\max_{u\in\mathcal{U}} Q^*(x,u)$$

We will estimate for (x,u) the optimal function $Q^*(x,u)$ by $Q^\pi(x,u)$, with π the best available policy. Hence, the best available action u^* will simply be

$$u^* = \arg\max_{u\in\mathcal{U}} Q^\pi(x,u)$$

The Q-learning algorithm aims at converging to the optimal Q-functions through iterations and updates of $Q^\pi(x,u)$ for all (x,u). The Q-values are stored into a Q-table in which the rows are the states x and the columns are the actions u. At the k^{th} step of the algorithm, given the learning rate α we perform the following substeps:

1) Choose action u_k in the space of possible actions given the agent's current state x_k.
2) Take the action and observe the new state $x_{k+1} = f(x_k, u_k)$, and the new space of possible actions $u_{k+1} \in \mathcal{U}_{x_{k+1}}$. The tuple $(x_k, u_k, x_{k+1} = f(x_k, u_k), r_k = r(x_k, u_k))$ is a transition.
3) Compute the maximum future expected reward for this transition: $r(x_k, u_k) + \gamma \max_{u_{k+1}\in\mathcal{U}_{x_{k+1}}} Q^\pi(f(x_k,u_k), u_{k+1})$.
4) Compute the error between the expected reward we actually have and the value stored in the Q-table for this transition:
$$r(x_k,u_k) + \gamma \max_{u_{k+1}\in\mathcal{U}_{x_{k+1}}} Q^\pi(f(x_k,u_k), u_{k+1}) - Q^\pi(x_k,u_k)$$
5) Update the Q-table and thus update the underlying policy π:
$$Q^\pi(x_k,u_k) \leftarrow Q^\pi(x_k,u_k) + \alpha\left[r(x_k,u_k) + \gamma \max_{u_{k+1}\in\mathcal{U}_{x_{k+1}}} Q^\pi(f(x_k,u_k), u_{k+1}) - Q^\pi(x_k,u_k)\right]$$

In general, to choose an action, we must use an *ϵ-greedy policy* to ensure that the agent will explore all the environment: with a probability ϵ, the agent chooses choose a random action, and with probability $1-\epsilon$, the agent will chooses the action with the maximum expected future reward, i.e. the best possible action given our current policy.

This algorithm works very well when the state and action spaces are small, but becomes inefficient as their size increases.The Q-table becomes to big to be fully updated in a reasonable time. Here, the solution we chose is deep Q-learning: instead of evaluating all Q-value, we try to approximate the Q-values for each action for a given state through a neural network which we will refer to as the Q-network.

Let θ represent the parameters of the Q-network. As we do not try to implement a policy, we will use $\hat{Q}(\cdot,\cdot,\theta) = Q^{\pi_\theta}(\cdot,\cdot)$ as the estimation of Q-functions.

At each step, we observe a sequence of transitions sampled following an ϵ-greedy policy. For each transition, the agent in state x takes an action u and goes to state $x' = f(x,u)$. Then we use the Q-network to estimate $\hat{Q}(x', u', \theta)$ for all possible u' and choose the action that yields the best possible expected future reward, and to estimate $\hat{Q}(x,u,\theta)$. The target Q-value has the same expression as in basic Q-learning: $r(x,u) + \gamma \max\limits_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta)$. We have all the values to compute the error for this transition (x,u,x'):

$$r(x,u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta) - \hat{Q}(x,u,\theta)$$

The real difference is that the algorithm will not directly update the Q-values. Instead, it updates the weights (or parameters, θ) of the Q-network. In order to do that, we will use the following loss function over the observation sequence [9]:

$$L_i(\theta_i) = \frac{1}{2}\mathbb{E}_{x,u,r,x'}\left[\left(r(x,u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta_i) - \hat{Q}(x,u,\theta_i)\right)^2\right] \tag{9}$$

This loss will be approximated at each step by the empirical estimation of the mean over a mini-batch of m transitions. Note the factor $\frac{1}{2}$ to simplify notations for the derivative.

Then the weights θ of the Q-network are updated, given learning rate α:

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta L_i(\theta)\,|_{\theta_i} \tag{10}$$

$$\theta_{i+1} = \theta_i + \alpha\left(r(x,u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta_i) - \hat{Q}(x,u,\theta_i)\right)\nabla_\theta \hat{Q}(x,u,\theta)\,|_{\theta_i} \tag{11}$$

However, as stated in Mnih et al, 2015 [9], *"Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q function)"*. There are at least three main reasons: first, Q-values are always modified, hence the data distribution and policy always change, leading to unstable targets. Second, target estimations and actual Q-value estimations are correlated as they are computed with the same Q-network and third, correlation in the observation sequence distribution. To address these issues, Mnih et al, 2015 [9] suggests two solutions:

- **Experience replay**: all transitions $(x_i, u_i, x'_i = f(x_i,u_i), r_i = r(x_i,u_i))$ are stacked into a list or array, the *replay memory*, from which we will sample some transitions to train the Q-network. The main improvements are (i) It reduces correlation over the sequence of transitions to better fit the i.i.d assumption needed for stochastic / mini-batch gradient-descent; (ii) smooths the data distribution (iii) avoid forgetting rare and useful experiences too rapidly.
- **Target Q-network**: we use a copy of the Q-network to compute the targets. This copy, named *Target-network* for clarity, with weights θ_i^-, is never used in training but is updated periodically, thus reducing the frequency of policy changes and reducing correlation between targets $r(x,u)+\gamma\hat{Q}(f(x,u),u',\theta_i)$ and actual values $\hat{Q}(x,u,\theta_i)$. This network will only be used yo compute target Q-values with older weights θ_i^-.

Thus when using experience replay, the sequence of observations used to train the Q-network is sampled from the replay memory. When using a Target-network, the loss function Equation 9 and weights updates Equation 11 must be fed with the following error:

$$r(x,u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta_i^-) - \hat{Q}(x,u,\theta_i) \tag{12}$$

Even with these two improvements, some inconsistencies remained in the way our algorithm works. We implemented 3 variations of the DQL Algorithm as an intent to correct them: Double Q-learning [5], Dueling Q-learning [16] and Prioritized Experience Replay (PER) [10]. Double Q-learning aims at reducing TD-targets overestimation by choosing the best next action in a different way. Dueling Q-network intends to improve the architecture of the Network by distinguishing between the Value of being in certain state and the Advantage given by taking a certain action. Prioritised Experience Replay is a means of storing and selecting transitions in the memory to facilitate training. Please refer to Appendix page 13 for more details. One of the objectives of our experiments on CartPole will be to measure the improvement they bring in terms of maximum expected reward and speed of convergence.

III. Implementation of the methods

For the Cart-Pole problem, the state space $\mathcal{X}$ is $\mathbb{R}^4$, and at step k the vector x_k summarizes the position and velocity of the cart, and the angle and angular velocity between the pole and the vertical. The action space $\mathcal{U}$ is just $\{0,1\}$, with 0 meaning that we apply a force of -1 to the cart (we push it left), and 1 meaning that we apply a force of $+1$ to the cart (we push it right). The game ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The dynamics and solution of such a game can be easily derived, but we will let the reinforcement learning algorithms do it for us.

In the case of Policy Gradient, if the neural network of parameters θ is a multi-layer perception, with an output layer of size $|\mathcal{U}|$, activated by a softmax function then we can easily process the log-likelihood. Let $\hat{y} = (\hat{y}_i)_{i=1,\dots,|\mathcal{U}|}$ be the output of the softmax activation layer, $\forall u_i \in \mathcal{U}, \hat{y}_i = \pi_\theta(u_i \mid x_k)$. Let y be the one hot encoded vector of the sampled action according to the probabilities given by $\hat{y}$ (if the sampled action is 0 for $\mathcal{U} = \{0, 1\}$, the one-hot encoded vector will be $(1, 0)$, otherwise, the one-hot encoded vector will be $(0, 1)$). In this case, the likelihood $\pi_\theta(u_k \mid x_k)$ of observing u_k (or y) given the current state x_k and the parameter θ is just $\prod_{u_i} \hat{y_i}^{y_i}$, and the log-likelihood will be $\sum_{u_i} y_i \log \hat{y}_i$.

By playing the n^{th} game entirely (or the n^{th} episode) with parameter θ_n in our neural network, let H be the step at which the game ends, we get a trajectory $\tau = (x_0, u_0, \dots, x_H, u_H)$. We also get a sequence of $(\hat{y}^0, y^0, \dots, \hat{y}^H, y^H)$ and a sequence of rewards $(r_0, \dots, r_H)$. Hence, at the end of the episode, we can process the update of the parameter θ with a learning rate $\alpha \in \mathbb{R}^+$,

$$\theta_{n+1} = \theta_n + \alpha \sum_k \gamma^k \left(\nabla_\theta \sum_{i=1}^{2} y_i^k \log \hat{y}_i^k \sum_{j=k}^{H} \gamma^{j-k} r_j \right)$$

Algorithm 1: Policy Gradient Algorithm

```
main initialization: Choose an initial parameter $\theta_0$, a learning rate $\alpha$ and a discount rate $\gamma$.
for $n = 0, \dots$ do
    state initialization: Get $x_0$ from the game, initialize $k$ to 0, the main trajectory $\tau$ to $()$,
    and the associated list of rewards $\rho$ to $()$
    while True do
        Sample $u_k \sim \pi_\theta(. \mid x_k)$
        $x_{k+1} \leftarrow f(x_k, u_k)$
        $r_k \leftarrow r(x_k, u_k)$
        Add $(x_k, u_k)$ to $\tau$
        Add $(r_k)$ to $\rho$
        if The game is finished then
            Break the loop
        end
        $k \leftarrow k + 1$
        $x_k \leftarrow x_{k+1}$
    end
    $\theta_{n+1} \leftarrow \theta_n + \alpha \nabla_\theta J(\theta)|_{\theta=\theta_n}$
end
```

In the case of Deep Q-learning, the neural network will have one dense hidden layer activated by ReLU and an size 2 output layer activated by a linear function (the Q-values may be negative).

As for policy gradient, by playing a game entirely with parameter θ in our neural network and parameter ϵ for our greedy policy, we arrive to and end at step H and get a trajectory $\tau = (x_0, u_0, ..., x_H)$. All transitions $(x_i, u_i, r_i, x_{i+1})_{0 \le i \le H-1}$ from this sequence will be added to the replay memory L_{replay}. If the size of the replay memory is large enough, the algorithm samples a minibatch, computes all actual Q-values and target Q-values from Target Q-network if Double Deep Q-learning is chosen and updates the DNN weights θ by optimizing. Every N_T optimization steps, the Target Q-network us updated. Algorithm 2 describes the way we apply Double Deep Q-learning to this problem.

For Dueling Q-learning, we changed the way the Q-network was build when we initiatize the agent. For prioritized Experience replay, we built a SumTree Binary tree class and a proper memory in the file *perutils.py* adapted from a code by Morvan Zhou [18].

IV. Experimentations and Results

We trained 5 different variations of Deep Q-Learning (DQL) that are Vanilla DQL, Double DQL, Dueling Double DQL, Dueling Double DQL with PER, and Double DQL with PER, and 3 different variations of Policy Gradient (PG) that are REINFORCE, A2C and PPO. We trained them all over 400 episodes[1] and tracked the score and the time taken to finish each episode to have a good vision of their convergence ability. For each episode, we also performed a test to obtain a score unbiased by the epsilon-greedy policy for DQL techniques. For all methods, we performed this training 20 times[1] in order to process a mean evolution the score and confidence intervals on this evolution. Finally, this training was done on the same machine to allow comparison. The hyperparameters used for each method can be found in Appendix. The code can be found on GitHub.

[1]Due to computational power issues, especially for Deep-Q-learning (~10-20min per training for 400 episodes), we could not do more than 20 trainings (computations) of 400 episodes per method/variation but we acknowledge that it would be necessary to have complete results.

Algorithm 2: Double Deep Q-Learning Algorithm (DDQN)

```
main initialization: Initial weights $\theta_0$, learning rate $\alpha$, discount rate $\gamma$, size of the minibatches $N_{batch}$, capacity of the replay memory $C_{replay}$, replay memory $L_{replay}$ to empty buffer of length $C_{replay}$, frequency of target network updates $N_{target}$, total number of optimization steps $N_{opti}$, greedy sequence $(\epsilon_k)_{k\in\mathbb{N}}$
for $n = 0, \dots$ do
    state initialization: Get $x_0$ from the game, initialize $k$ to 0
    while True do
        With probability $\epsilon_k$ select a random action $u_k$
        Otherwise select $u_k = \arg\max_{u\in\mathcal{U}_{x_k}} \hat{Q}(x_k, u, \theta_k)$
        $x_{k+1} \leftarrow f(x_k, u_k)$, $r_k \leftarrow r(x_k, u_k)$
        if $len(L_{replay}) \geq C_{replay}$ then
            Remove oldest transition from $L_{replay}$
        end
        Add transition $(x_k, u_k, x_{k+1}, r_k)$ to $L_{replay}$
        if $len(L_{replay}) > N_{batch}$ then
            Sample random minibatch of $N_{batch}$ transitions $(x_k^i, u_k^i, {x_k^i}', r_k^i)_{1\leq i\leq N_{batch}}$ ~Unif($L_{replay}$)
            Compute targets: $t_k^i = \begin{cases} r_k^i \text{ if } {x_k^i}' \text{ is the termination state} \\ r_k^i + \hat{Q}({x_k^i}', \arg\max_{u'\in\mathcal{U}_{{x_k^i}'}} \{\hat{Q}({x_k^i}', u', \theta_k)\}, \theta_k^-) \text{ otherwise} \end{cases}$
            Compute Q-values estimations $\hat{Q}(x_k^i, u_k^i, \theta_k)$
            Compute loss $L_k(\theta_k) \leftarrow \frac{1}{2N_{batch}} \sum_{i=1}^{N_{batch}} (t_k^i - \hat{Q}(x_k^i, u_k^i, \theta_k))^2$
            Perform gradient descent: $\theta_{k+1} \leftarrow \theta_k + \alpha\nabla_\theta L_k(\theta)|_{\theta=\theta_k}$
            $N_{opti} \leftarrow N_{opti} + 1$
            if $N_{opti} \% N_{target}$ = 0 then
                Update target network: $\theta_k^- = \theta_k$
            end
        end
        if $x_{k+1}$= terminated then
            Break the loop
        end
        $k \leftarrow k + 1$
        $x_k \leftarrow x_{k+1}$
    end
end
```

On Figure 2, one can visualise the mean evolution of the score over episodes with 90% confidence intervals. The initial data being particularly noisy, we decided to plot the rolling average score processed with a 50 episodes window instead of the raw score.

- Regarding DQL methods, on Cart-Pole the results are almost the same whatever the variation: a steady and rapid improvement in score from episodes 50 to 200, to reach more than 190. Double DQL increases the speed of convergence (local maximum for the score around 180 episodes against 250 for Vanilla DQL). Dueling allows to have slightly better confidence intervals: the convergence is more stable. PER converges faster (around 150 episodes) on average and helps stabilize the score around the maximum value.
- On the other hand, the large confidence intervals of PG methods show that their performances are much less consistent, especially for REINFORCE with performances ranging from worst to best. This is in fact a huge issue of Policy Gradient methods, the convergence is poor due to highly biased gradients as we train the model on a dataset built with a single policy (which is not the case for DQL). A2C aims at solving this problem by introducing a baseline in the gradient, and we clearly see an improvement in the confidence interval for this method compare to REINFORCE. However, its convergence is far from perfect, probably due to abrupt policy changes from one episode to the next. PPO was invented to avoid these abrupt changes and increase the training performances. Indeed, the figure shows an ever better confidence interval than for other PG methods with final scores higher than those of REINFORCE and A2C on average.

On Figure 3, it seems that DQL methods outperform Policy Gradient ones by far. However, it is important to keep in mind that due to the architecture of the algorithms, DQL methods achieve as many optimization step as there are game steps per episode, whereas PG methods achieve only one optimization step per episode. Therefore comparing them on an episode scale gives only a partial answer and other elements need to be studied in order to give a proper conclusion. We will therefore compare them on some other criteria. The first one is the quality of the convergence: looking at the confidence intervals and mean evolution on Figure 2, one can clearly observe that DQL methods are more robust than PG methods; they tend to reach a ”more global” optimum fairly soon, whereas PG methods may fall in many local optima before reaching ”more global” optima, if they reach it. In other words,

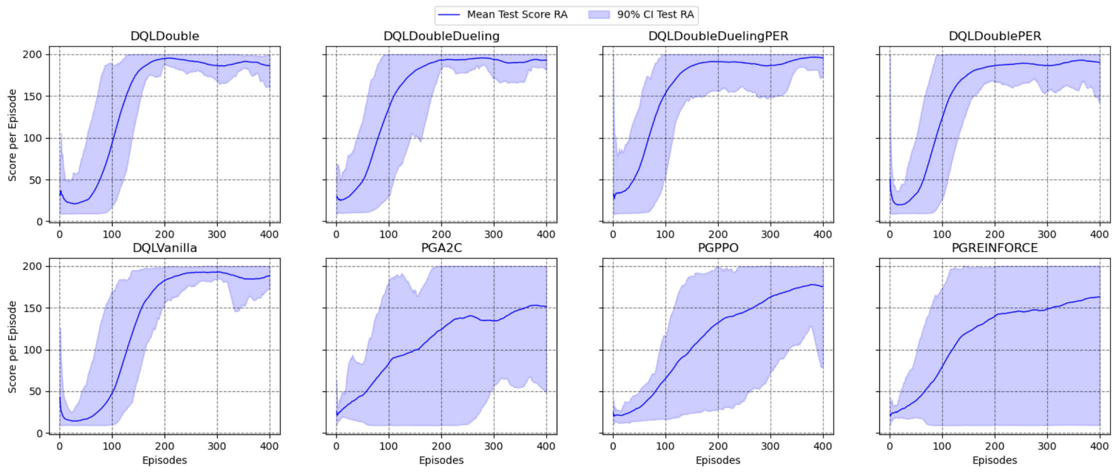

Fig. 2: Evolution of the rolling average score earned on the test phase during the training for each method. We ran one test episode after each training episode to eliminate the bias introduced by the ϵ-greedy policy in action choice. The rolling average is processed over a window of length 50 episodes. RA stands for rolling average. The means and 90% confidence intervals are processed over 20 computations.

the probability of convergence towards the global minimum (i.e. a policy almost always leading to scores close to the maximum) is higher for DQL than for PG methods.

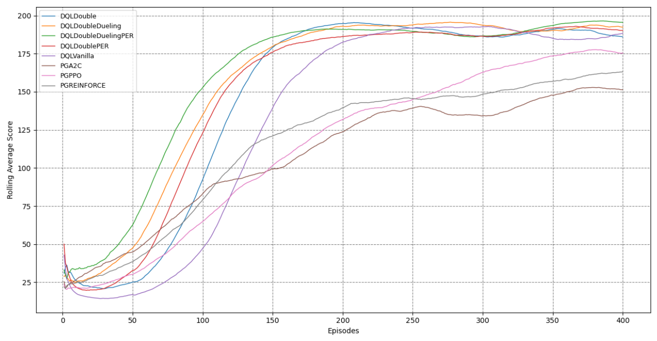

Fig. 3: Comparison of all methods on the mean evolution of the rolling average score earned on the testing phase during the training. Rolling averages are processed on a sliding window of 50 episodes. The mean evolutions are processed over 20 computations.

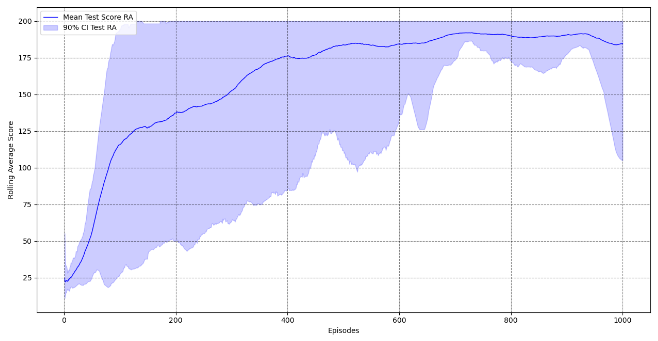

Fig. 4: Evolution of the rolling average score earned on the testing phase during a training over 1000 episodes for A2C, with 90% Confidence intervals (∼3h of training on the machine we used). The rolling average is processed on a window of length 50 episodes. A2C converges in more than 500 episodes.

The second criterion is the time cost of both methods. Table I summarizes the average time taken by each method to complete one episode (a game). PG methods are about 10 times faster on average than DQL methods. This can largely be explained by the number of optimization steps for each type of method, as this is clearly the most time-consuming task during training. Moreover, it is important to note that since the DQL curves took off faster, the number of game steps per episode on average is higher than for the PG methods and therefore increases the duration of the episodes.

TABLE I: Time in seconds took on average by each method to realize one episode (Average processed over 20 computations).

Method	Average Time per episode (s)	Change in Avg Time compared to ref (=0.0%)	Minimum Average Time (s)	Maximum Average Time (s)	Standard Deviation of Average Time (s)
DQL Vanilla	1.32	-29.0%	1.15	1.72	0.13
DQL Double	1.86	0.0 %	1.57	2.07	0.15
DQL Double + Dueling	3.32	78.5%	2.81	4.07	0.34
DQL Double + Dueling + PER	4.07	118.8%	2.98	4.71	0.38
DQL Double + PER	2.50	34.4%	2.04	2.88	0.22
PG REINFORCE	0.22	0.0%	0.02	0.35	0.09
PG A2C	0.20	-10.1%	0.02	0.30	0.07
PG PPO	0.18	-18.0%	0.08	0.23	0.05

Finally, we looked at the first time each method mean evolution would reach and exceed a specific score. We retained 180 (out of 200) as a good metric. The comparison of when all methods exceeded 180 is shown in Figure 5 on a logarithmic scale. PG methods are clearly an order of magnitude faster than DQL methods on average. Focusing on DQL variations and comparing them to Double DQL, the Dueling architecture has multiplied training time by 1.78 on average, and PER has increased it by 1.34. However when using Dueling architecture and PER, we see an increase of 2.19 instead of an expected $2.39 = 1.34 \times 1.78$: we are gaining a bit by combining Dueling and PER. For PG variations, we see a decrease in time compared to the Vanilla ”REINFORCE” method. As the same amount of processing is needed for each method, we can explain this difference mainly by the lower number of game steps performed per episode on average for PPO and A2C over these 400 episodes (both curves took longer to take off than the REINFORCE curve, see Figure 3).

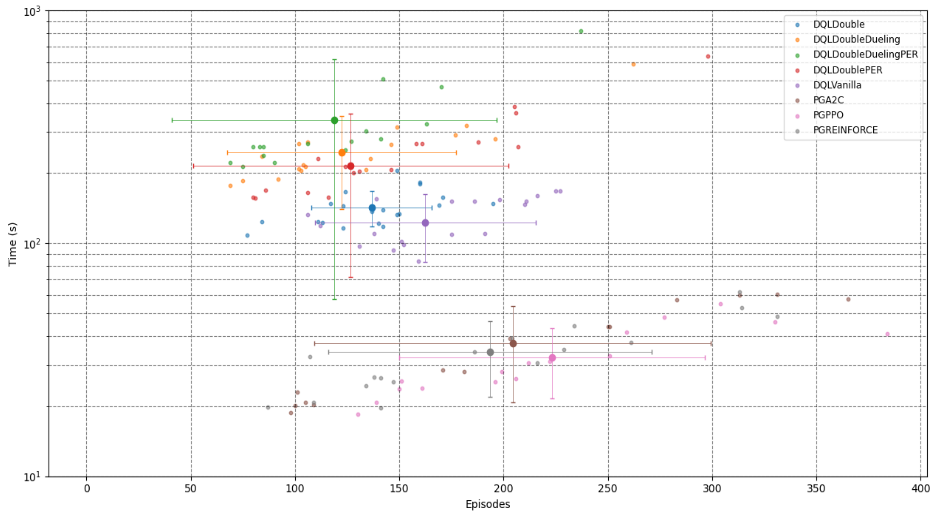

Fig. 5: Comparison of all methods: first time that the rolling average score exceeded 180 during the test phase on all trainings that reached 180 out of 20 trainings per method. The rolling average is processed on a window of length 50 episodes. The average point standard deviation for each method is displayed.

V. Conclusion

This project aimed to identify differences between Policy Gradient method and Deep Q-learning, as well as some improvements for both algorithms. Based on our analysis, Deep Q-Learning is more stable and more robust but Policy gradient converges faster over time. PG improvements we chose (A2C, PPO) reduce the uncertainty about the time required to converge. For Deep Q-learning, Double Q-Learning is a great improvement to converge in less episodes with almost no increase in computing time. Although Prioritized experience replay and Dueling architecture stabilize the learning and accelerate it in terms of episodes, it is not efficient in terms of training time. Unfortunately due to the low complexity of the Cart-pole problem, we could not link any increase or decrease in the final average score to the choice of methods and variations: they all reached scores above 190 at a point. A last advantage of PG methods over DQL methods is the small number of hyperparameters to be tuned. While this is clearly not an issue for a problem as easy as the Cart-Pole, it becomes infinitely important to actively tune the hyperparameters for more complex problem, and methods such as PG greatly simplify this process.

We believe that a good compromise would be to combine Policy Gradient and Deep Q-learning for more complex problems: Policy Gradient for on-policy learning and Q-learning for off-policy learning. The computing time would then be reduced but we would not fall into a local minimum for the loss function thanks to DQL. A similar algorithm called Deep Deterministic Policy Gradient was used by DeepMind to build and train AlphaGo, the famous algorithm that defeated the world Go champion in 2016 [14].

Unfortunately we lacked computational power to complete all the work we wanted to do, but it can be seen as potential enhancements of this research to measure the exact benefit provided by each method in terms of performance, training time, and training stability:

- Do our experiments with more computations and episodes: ideally on cart-pole 50 computations, 1000 episodes.
- Experiment on Breakout and other games from Atari (100000+ episodes needed) with sparser rewards.

References

[1] Richard Bellman. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[2] Thomas Degris, Martha White, and Richard S. Sutton. “Off-Policy Actor-Critic”. In: *CoRR* abs/1205.4839 (2012). arXiv: 1205.4839.

[3] Tuomas Haarnoja et al. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290.

[4] Hado V. Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621.

[5] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, 2016, 2094–2100.

[6] Gareth James et al. *An Introduction to Statistical Learning with Applications in R*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2017.

[7] Jingbin Liu et al. “On-policy Reinforcement Learning with Entropy Regularization”. In: *arXiv preprint arXiv:1912.01557* (2019).

[8] Adventures in Machine Learning. *SumTree introduction in Python*. [Online; accessed 21-April-2020]. 2020.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (2015), pp. 529–533.

[10] Tom Schaul et al. “Prioritized Experience Replay”. In: *CoRR* abs/1511.05952 (2016). arXiv: 1511.05952.

[11] John Schulman et al. “High-dimensional continuous control using generalized advantage estimation”. In: *arXiv preprint arXiv:1506.02438* (2015).

[12] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347.

[13] John Schulman et al. “Trust Region Policy Optimization”. In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477.

[14] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016), pp. 484–489.

[15] Richard S Sutton et al. “Policy gradient methods for reinforcement learning with function approximation”. In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[16] Ziyu Wang et al. “Dueling Network Architectures for Deep Reinforcement Learning”. In: (2015). arXiv: 1511.06581.

[17] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[18] Morvan Zhou. *The DQN improvement: Prioritized Experience Replay*. [Online; accessed 13-April-2020, commit on master 5-August-2018]. 2018.

Appendix

Appendix 1: Hyperparameters

TABLE II: Hyperparameters and Architecture for Deep Q-Learning

Variations	Vanilla	Double	Double + Dueling	Double + PER	Double + Dueling + PER
1D Conv layers	None	None	None	None	None
Dense layers	1x 64	1x 64	1x 32	1x 64	1x 32
Number of parameters	450	450	419	450	419
Initializer			He initializer		
Optimizer			Default Adam from Tensorflow		
Size of minibatch	32	32	32	32	32
Discount rate γ	0.99	0.99	0.99	0.99	0.99
Learning rate α	1e-3	1e-3	1e-3	1e-3	1e-3
Greedy policy ϵ			$\epsilon_{start} = 1$, $\epsilon_{end} = 0.1$, $N_{decaysteps} = 1000$		
N_{target}[2]	200	200	200	200	200
Replay memory capacity	20 000	20 000	20 000	20 000	20 000
PER Hyperparamaters [3]					
α				0.6	0.6
β_{start}				0.4	0.4
β_{end}				1	1
$\beta_{increment}$				1e-4	1e-4
Minimum proba ϵ				1e-3	1e-3

TABLE III: Hyperparameters and Architecture for Policy Gradients

Variations	REINFORCE	A2C	PPO
1D Conv layers	None	None	None
Dense layers Actor	64	64	64
Number of parameters Actor	450	450	450
Dense layers Critic		64	64
Number of parameters Critic		385	385
Initializer		Random normal initializer	
Optimizer		Default Adam from Tensorflow	
Discount rate γ	0.99	0.99	0.99
Actor Learning Rate	5e-3	5e-3	5e-3
Critic Learning Rate		5e-3	5e-3
PPO Parameter ϵ			0.2

Appendix 2: Policy Gradient Advanced Methods: General Advantage Estimate (GAE), Proximal Policy Optimization (PPO) and Entropy Regularization

General Advantage Estimate

In [11], a more general version of the advantage is derived. General Advantage Estimation aims at generalizing this principle of action advantage. They define the TD residual associated to the approximated value function $V_\psi^{\pi_\theta}$ at each step k by

$$\delta_k^{V_\psi^{\pi_\theta}} = r(x_k, u_k) + \gamma V_\psi^{\pi_\theta}(x_{k+1}) - V_\psi^{\pi_\theta}(x_k)$$

Using these residuals, they define for some $\lambda \in [0, 1]$ the general advantage estimate at step k:

$$A_\psi^{\pi_\theta\, GAE} = \sum_{l \geq k} (\gamma\lambda)^{k-l} \delta_k^{V_\psi^{\pi_\theta}}$$

This definition of the advantage allows us to find a trade off between bias and variance. When λ tends to 1, the bias increases and the variance decreases until $\lambda = 1$, where the general advantage estimate will be equivalent to the one defined in Equation 8. When λ tends to 0, the contrary happens.

[2]Target-network weights are updated every N_{target} optimization steps. See algorithm 2 for more details.
[3]Adapted directly from hyperparameters given by [10].

Proximal Policy Optimization

Even with an optimal advantage estimate, brutal changes in the policy may still happen. A solution developed in [12] puts an end to this problem. *Proximal Policy Optimization* (or PPO) method aims at avoiding these brutal changes by limiting their momentum. We define the probability ratio $r_k(\theta)$ at step k:

$$r_k(\theta) = \frac{\pi_\theta(u_k \mid x_k)}{\pi_{\theta_{old}}(u_k \mid x_k)}$$

With θ_{old} a parameter of the Actor network used previously (in practice we take the one used to generate trajectories used during the last optimization step). PPO is based on the fact that solving $\arg\max_\theta J(\theta)$ is equivalent to solve $\arg\max_\theta J(\theta) - J(\theta_{old})$.

$$J(\theta) - J(\theta_{old}) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_k \gamma^k A^{\pi_{\theta_{old}}}(x_k, u_k)\right] \qquad \textit{Proof in Appendix} \quad (13)$$

Consider the discounted future state distribution

$$d^{\pi_\theta}(x) = (1-\gamma)\sum_k \gamma^k P(x_k \mid \pi_\theta) \qquad \textit{Proof in Appendix} \quad (14)$$

We can rewrite the difference $J(\theta) - J(\theta_{old})$ using the form from [15] and the fact that a known distribution can be used to estimate another distribution:

$$J(\theta) - J(\theta_{old}) = \mathbb{E}_{x\sim d^{\pi_\theta}}\left[\mathbb{E}_{u\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(u \mid x)}{\pi_{\theta_{old}}(u \mid x)} A^{\pi_{\theta_{old}}}(x,u) \;\middle|\; x\right]\right] \qquad \textit{Proof in Appendix} \quad (15)$$

If at each update of θ we ensure that π_θ is not that different from $\pi_{\theta_{old}}$ by maintaining $r_j(\theta) \approx 1$, for example by clipping its value between $1-\epsilon$ and $1+\epsilon$ for $\epsilon > 0$ small enough, we can suppose that $d^{\pi_\theta} \approx d^{\pi_{\theta_{old}}}$ and:

$$\begin{aligned} J(\theta) - J(\theta_{old}) &\approx \mathbb{E}_{x\sim d^{\pi_{\theta_{old}}}}\left[\mathbb{E}_{u\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(u \mid x)}{\pi_{\theta_{old}}(u \mid x)} A^{\pi_{\theta_{old}}}(x,u) \;\middle|\; x\right]\right] \\ &= \mathbb{E}_{\tau\sim\pi_{\theta_{old}}}\left[\sum_k \gamma^k r_k(\theta) A^{\pi_{\theta_{old}}}(x_k, u_k)\right] \end{aligned}$$

And the gradient becomes (with the approximation of $A^{\pi_{\theta_{old}}}$ parameterized by ψ):

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau\sim\pi_{\theta_{old}}}\left[\sum_k \gamma^k \nabla_\theta r_k(\theta) A_\psi^{\pi_{\theta_{old}}}(x_k, u_k)\right]$$

Maintaining $r_j(\theta) \approx 1$, that's exactly what PPO is doing by considering the surrogate objective function to optimize

$$L(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta_{old}}}\left[\sum_k \gamma^k \min\left\{r_k(\theta) A_\psi^{\pi_{\theta_{old}}}(x_k, u_k), \operatorname{clip}\left(r_k(\theta), 1-\epsilon, 1+\epsilon\right) A_\psi^{\pi_{\theta_{old}}}(x_k, u_k)\right\}\right]$$

Entropy Regularization

By limiting the change in policy from one update step to the next, the exploration ability of the method might become very low. The trick to avoid this issue is to introduce an entropy regularization term in the loss function. But first, we need to make a distinction between on-policy and off-policy methods. While the first one trains neural networks on trajectories sampled with the current policy we are updating, the second one keeps sampled trajectories from all past policies in memory and trains neural networks on batches of trajectories drawn from his memory. The first off-policy gradient algorithm was developed in [2]. As trajectories in the memory follows policies different from the current policy, it has proven efficiency in exploration. All methods treated in this paper are on-policy methods.

Coming back to our entropy term to include in the loss function, while this might be tricky to derive for off-policy methods such as *Soft Actor Critic* [3] that uses an entropy term based on the Kullback-Leiber divergence with *Trust Region Policy Optimization* paradigm [13], this can also be derived for on-policy methods such as PPO [7]. To do so, we need to first define the entropy of the policy π_θ at step k for a given state x_k:

$$H: \quad \pi_\theta, x_k \to \mathbb{E}_{u_k}\left[-\log \pi_\theta(u_k \mid x_k) \mid x_k\right]$$

Let $t \in \mathbb{R}^*$ be the temperature parameter, which translate the importance we give to the entropy during the learning, we introduce this entropy term in both the Value and the Q-value function:

$$\begin{aligned}\tilde{Q}^\pi : \quad & \mathcal{X} \times \mathcal{U} \to \mathbb{R} \\ & x_k, u_k \to \mathbb{E}_{\substack{x_{k+1:} \\ u_{k+1:}} \sim \pi} \left[\sum_{j \geq k} \gamma^{j-k} \Big(r(x_j, u_j) + tH(\pi_\theta, x_k) \Big) \mid \pi, x_k, u_k \right] \\ \tilde{V}^\pi : \quad & \mathcal{X} \to \mathbb{R} \\ & x_k \to \mathbb{E}_{\substack{x_{k+1:} \\ u_{k:}} \sim \pi} \left[\sum_{j \geq k} \gamma^{j-k} \Big(r(x_j, u_j) + tH(\pi_\theta, x_k) \Big) \mid \pi, x_k \right]\end{aligned}$$

And [7] demonstrates that

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) \tilde{A}_\psi^{\pi_\theta}(x_k, u_k) \right]$$

With $\tilde{A}_\psi^{\pi_\theta}(x_k, u_k) = A_\psi^{\pi_\theta}(x_k, u_k) - t \log \pi_\theta(u_k \mid x_k)$. Hence we can easily derive $L(\theta)$ used for PPO:

$$L(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \left[\sum_k \gamma^k \min \left\{ r_k(\theta) \tilde{A}_\psi^{\pi_{\theta_{old}}}(x_k, u_k), \text{clip}\left(r_k(\theta), 1-\epsilon, 1+\epsilon\right) \tilde{A}_\psi^{\pi_{\theta_{old}}}(x_k, u_k) \right\} \right]$$

Appendix 3: Deep Q-learning Variations: Double DQL, Dueling DQL, Prioritized Experience replay

Double Q-learning

As stated in Hasselt et al, 2010 [4], the Q-learning algorithm and Deep Q-learning are overestimating TD-targets at all steps k by using $\max_{u'} \hat{Q}(x'_k, u', \theta_k)$ to estimate $\mathbb{E}_{x'_k \in \mathcal{X}}[\max_{u'} \hat{Q}(x'_k, u', \theta_k)]$ which is an approximation for $\max_{u'} \left\{ \mathbb{E}_{x'_k \in \mathcal{X}} \left[\hat{Q}(x'_k, u', \theta_k) \right] \right\}$, where $x'_k = f(x_k, u_k)$ for a transition (x_k, u_k, x'_k, r_k).

In order to decrease Q-values overestimations, Hasselt et al, 2016 suggests decomposing "*the max operation in the target into action selection and action evalutation*" [5]. Instead of selecting the best action by taking the best Q-value $\max_{u'} \hat{Q}(f(x_k, u_k), u', \theta_k^-)$, we will select the best action according to the Q-network $\arg\max_{u'}\{\hat{Q}(f(x_k, u_k), u', \theta_k)$ and compute the related Q-value with the Target-network using θ_k^-. The error at step k will then be:

$$r(x_k, u_k) + \gamma \hat{Q}(f(x_k, u_k), \arg\max_{u'}\{\hat{Q}(f(x_k, u_k), u', \theta_k)\}, \theta_k^-) - \hat{Q}(x_k, u_k, \theta_k)$$

Dueling Q-learning

Developed in Wang et al, 20165 [16], the dueling network exploits the difference between the value of being in one state and the comparative advantage of an action over others in order to determine the Q-values. We stick with Q^π and V^π as defined in Equation 1 and Equation 2. Then we define the advantage function taking action u_k given that we are currently in state x_k, following policy π_θ, as in Equation 3:

$$\begin{aligned} A^{\pi_\theta} : \quad & \mathcal{X} \times \mathcal{U} \to \mathbb{R} \\ & (x_k, u_k) \to Q^{\pi_\theta}(x_k, u_k, \theta_k) - V^{\pi_\theta}(x_k) \end{aligned}$$

As we did with $Q^{\pi_\theta} = \hat{Q}(\cdot, \cdot, \theta)$, we will continue with $V^{\pi_\theta} = \hat{V}(\cdot, \theta)$ and $A^{\pi_\theta} = \hat{A}(\cdot, \cdot, \theta)$.

Wang et al tried to divide the end of the Deep Q-network in *two sequences (or streams) of fully connected layers. The streams are built such that they have the capability of providing separate estimates of the value and advantage functions*. [16]. The two streams take as input the output of the common convolutional layers. Note that the output of the Value stream has only one node for the current state, whereas the output for the Advantage stream has as many nodes as they are actions in $\mathcal{U}$. Then the two streams are combined using a special aggregation layer to get an estimate of Q-values. Let θ be the parameters for the common layers, α be the parameters for the Advantage stream, and β for the Value stream. Then after an aggregation module, we have:

$$\hat{Q}(x, u, \theta, \alpha, \beta) = \hat{V}(x, \theta, \beta) + \hat{A}(x, u, \theta, \alpha)$$

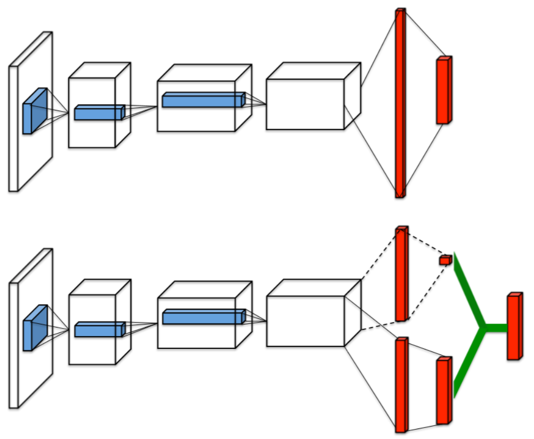

Fig. 6: Picture describing a popular deep-Q network architecture (top) and the dueling network architecture (bottom) from [16]. the green line is the combination module used to compute Q-values from Values and Advantages.

As stated by Wang et al, we are unable to retrieve $\hat{A}$ and $\hat{V}$ values from $\hat{Q}$ values. This is an issue of identifiability that they solved by deviating from the original definition of $\hat{V}$ and $\hat{A}$. They forced the advantage part of the Q-value to have 0 mean by taking this formula to compute the final Q-value:

$$\hat{Q}(x,u,\theta,\alpha,\beta) = \hat{V}(x,\theta,\beta) + \left(\hat{A}(x,u,\theta,\alpha) - \frac{1}{|\mathcal{U}|}\sum_{u'\in\mathcal{U}}\hat{A}(x,u',\theta,\alpha)\right)$$

Why do that ? It allows to decouple the state value from the action value and thus learn what states are valuable, and if taking an action will really change the outcome in a given situation (In a car driving game, moving right or left is important only if there is a risk of collision). All other parts of the network and training are identical to the baseline DQL algorithm and can be improved using other methods. Dueling is only a special architecture for the network. We used in our implementation a single fully connected layer for both the Value stream and the Advantage stream.

Prioritized Experience Replay

Introduced (also) by GoogleDeepMind in 2016 in [10], *Prioritized Experience Replay* main assumption is that some experiences are more important and effective in the agent's learning process, and that it may change over the course of the learning: some transitions might become increasingly helpful for the learning as the agent accuracy increases. To go from uniform replay to prioritized replay, we need to associate a priority score to each transition stored in the replay memory. Schaul et al, 2015 [10] suggests to use the TD-error δ as a basis. Let transition i be $(x_i, u_i, x'_i = f(x_i,u_i), r_i = r(x_i,u_i))$ with error δ_i sampled from the replay memory containing N_{replay} experiences. This paper define the priority score of transition i as:

$$p_i = |\delta_i| + \epsilon \quad \text{with } \epsilon > 0$$

Then the probability of sampling transition i from the replay memory is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_{j=1}^{N_{replay}} p_j^\alpha} \tag{16}$$

The $\epsilon > 0$ constant prevents cases of having transitions with a 0 probability once their error is 0. The hyperparameter $\alpha \in [0,1]$ determines the prioritization intensity, where $\alpha = 0$ being the standard uniform replay memory.

Two options are considered in the paper: rank-based or proportional prioritization. We will only consider proportional prioritization here, meaning that we will sample transitions using the probability $P(i)$ and not $\frac{1}{rank(i)}$ with the rank of the transition in the replay memory given by the priority score p_i.

Estimating the expected value of a statistic with Stochactic and mini-batch Gradient Descent relies on having the same distribution in samples as in the reality, so that Gradient Descent updates accurately reflect the distribution of real data [6]. However, *"prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to"* [10]. The paper introduces Importance-Sampling (IS) weights to compensate:

$$w_i = \left(\frac{1}{N_{replay}} \cdot \frac{1}{P(i)}\right)^\beta \tag{17}$$

where $\beta \in [0,1]$ is the compensation hyperparameter. For $\beta = 1$ bias introduced by prioritization is fully cancelled in the transitions distribution. The IS weights are always normalized with $\frac{1}{\max_j w_j}$ for stability reasons [10].

Thus at step k for a minibatch of size m where all transitions $(x_i, u_i, x'_i = f(x_i,u_i), r_i = r(x_i,u_i))$ are sampled according to $P(i)$, the update becomes:

$$\begin{aligned}\theta_{k+1} &= \theta_k + \alpha \nabla_\theta L_k(\theta)\,|_{\theta_k} \\ &= \theta_k + \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}\frac{w_i}{\max\limits_{1\le j\le N_{replay}} w_j}\left(r_i + \gamma \max_{u'\in\mathcal{U}_{x'_i}}\hat{Q}(x'_i,u',\theta_k^-) - \hat{Q}(x_i,u_i,\theta_k)\right)\nabla_\theta\hat{Q}(x_i,u_i,\theta)\,|_{\theta_k}\end{aligned} \tag{18}$$

The implementation of Prioritized Experience Replay requires a very efficient way of storing, accessing and updating transitions in the replay memory. We followed the recommendation of Schaul et al, 2015 [10] by using a sum-tree data structure. It is a binary tree (each parent has no more than two children) where the parents value are the sum of the children values. A very good explanation can be found in here [8]. For prioritization, the leaf node value is the probability $P(i)$, and the index of the node is used to retrieve the related data in the tree. We used the implementation by Morvan Zhou from [18] as a strong basis for our code.

Appendix 4: Policy Gradient Equations Proofs

Demonstration Equation 4:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau\in\mathfrak{T}} \pi_\theta(\tau) R(\tau) \\
&= \sum_{\tau\in\mathfrak{T}} \nabla_\theta \pi_\theta(\tau) R(\tau) \\
&= \sum_{\tau\in\mathfrak{T}} \frac{\pi_\theta(\tau)}{\pi_\theta(\tau)} \nabla_\theta \pi_\theta(\tau) R(\tau) \\
&= \sum_{\tau\in\mathfrak{T}} \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} R(\tau) \\
&= \sum_{\tau\in\mathfrak{T}} \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) R(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(\tau) R(\tau)\right]
\end{aligned}$$

Demonstration Equation 5:

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(\tau) &= \nabla_\theta \log\left(P(x_0)\prod_k \pi_\theta(u_k \mid x_k) P(x_{k+1} \mid x_k, u_k)\right) \\
&= \nabla_\theta \log P(x_0) + \nabla_\theta \log\left(\prod_k \pi_\theta(u_k \mid x_k) P(x_{k+1} \mid x_k, u_k)\right) \\
&= \sum_k \nabla_\theta \log\left(\pi_\theta(u_k \mid x_k) P(x_{k+1} \mid x_k, u_k)\right) \\
&= \sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) + \nabla_\theta \log P(x_{k+1} \mid x_k, u_k) \\
&= \sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k)
\end{aligned}$$

Demonstration Equation 6:

$$\begin{aligned}
\mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) b \mid \pi_\theta\right] &= \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k-1}}\sim\pi_\theta}\left[\mathbb{E}_{\substack{x_{k+1:} \\ u_{k:}}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) b \mid x_0, u_0, \dots, u_{k-1}, x_k\right]\right] \\
&= \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k-1}}\sim\pi_\theta}\left[b\mathbb{E}_{\substack{x_{k+1:} \\ u_{k:}}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) \mid x_0, u_0, \dots, u_{k-1}, x_k\right]\right] \\
&= \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k-1}}\sim\pi_\theta}\left[b\mathbb{E}_{u_k\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) \mid x_k\right]\right] \\
&= \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k-1}}\sim\pi_\theta}\left[b\mathbb{E}_{u_k\sim\pi_\theta}\left[\frac{\nabla_\theta \pi_\theta(u_k \mid x_k)}{\pi_\theta(u_k \mid x_k)} \mid x_k\right]\right] \\
&= \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k-1}}\sim\pi_\theta}\left[b\nabla_\theta\mathbb{E}_{u_k\sim\pi_\theta}\left[1 \mid x_k\right]\right] \\
&= 0
\end{aligned}$$

Demonstration Equation 7:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k)\left(\sum_{j\geq k} r(x_j, u_j)\right)\right] \\
&= \sum_k \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k}}\sim\pi_\theta}\left[\mathbb{E}_{\substack{x_{k+1:} \\ u_{k+1:}}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(u_k \mid x_k)\left(\sum_{j\geq k} r(x_j, u_j)\right) \mid x_0, u_0, \dots, x_k, u_k\right]\right]
\end{aligned}$$

$$
\begin{aligned}
&= \sum_k \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k}} \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) \mathbb{E}_{\substack{x_{k+1:} \\ u_{k+1:}} \sim \pi_\theta} \left[\sum_{j \geq k} r(x_j, u_j) \mid x_0, u_0, \dots, x_k, u_k \right] \right] \\
&= \sum_k \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k}} \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) \mathbb{E}_{\substack{x_{k+1:} \\ u_{k+1:}} \sim \pi_\theta} \left[\sum_{j \geq k} r(x_j, u_j) \mid x_k, u_k \right] \right] \\
&= \sum_k \mathbb{E}_{\substack{x_{0:k} \\ u_{0:k}} \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k) \right]
\end{aligned}
$$

Passage from row 3 to row 4 is possible because we are dealing with 1^{st} order Markov Chains.
Demonstration Equation 13:

$$
\begin{aligned}
J(\theta) - J(\theta_{old}) &= J(\theta) - \mathbb{E}_{\tau \sim \pi_\theta} \left[V^{\pi_{\theta_{old}}}(x_0) \right] \\
&= J(\theta) + \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{k \geq 1} \gamma^k V^{\pi_{\theta_{old}}}(x_k) - \sum_{k \geq 0} \gamma^k V^{\pi_{\theta_{old}}}(x_k) \right] \\
&= J(\theta) + \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \gamma^{k+1} V^{\pi_{\theta_{old}}}(x_{k+1}) - \gamma^k V^{\pi_{\theta_{old}}}(x_k) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \gamma^k \Big(r(x_k, u_k) + \gamma V^{\pi_{\theta_{old}}}(x_{k+1}) - V^{\pi_{\theta_{old}}}(x_k) \Big) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \gamma^k \Big(Q^{\pi_{\theta_{old}}}(x_k, u_k) - V^{\pi_{\theta_{old}}}(x_k) \Big) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \gamma^k A^{\pi_{\theta_{old}}}(x_k, u_k) \right]
\end{aligned}
$$

Demonstration Equation 14: It is indeed a distribution in the hypothesis that the trajectory is infinite:

$$
\begin{aligned}
\sum_{x \in \mathcal{X}} d^{\pi_\theta}(x) &= \sum_{x \in \mathcal{X}} (1-\gamma) \sum_k \gamma^k P(x = x_k \mid \pi_\theta) \\
&= (1-\gamma) \sum_k \gamma^k \sum_{x \in \mathcal{X}} P(x = x_k \mid \pi_\theta) \\
&= (1-\gamma) \sum_k \gamma^k \\
&= \frac{1-\gamma}{1-\gamma} \\
&= 1
\end{aligned}
$$

Demonstration Equation 15:

$$
\begin{aligned}
J(\theta) - J(\theta_{old}) &= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_k \gamma^k A^{\pi_{\theta_{old}}}(x_k, u_k) \right] \\
&= \mathbb{E}_{x \sim d^{\pi_\theta}} \left[\mathbb{E}_{u \sim \pi_\theta} \left[A^{\pi_{\theta_{old}}}(x, u) \mid x \right] \right] \\
&= \mathbb{E}_{x \sim d^{\pi_\theta}} \left[\sum_{u \in \mathcal{U}} \pi_\theta(u \mid x) A^{\pi_{\theta_{old}}}(x, u) \right] \\
&= \mathbb{E}_{x \sim d^{\pi_\theta}} \left[\sum_{u \in \mathcal{U}} \pi_{\theta_{old}}(u \mid x) \frac{\pi_\theta(u \mid x)}{\pi_{\theta_{old}}(u \mid x)} A^{\pi_{\theta_{old}}}(x, u) \right] \\
&= \mathbb{E}_{x \sim d^{\pi_\theta}} \left[\mathbb{E}_{u \sim \pi_{\theta_{old}}} \left[\frac{\pi_\theta(u \mid x)}{\pi_{\theta_{old}}(u \mid x)} A^{\pi_{\theta_{old}}}(x, u) \mid x \right] \right]
\end{aligned}
$$