On the use of Reinforcement Learning to play Atari Games

IEOR 265, University of California, Berkeley

(Milestone Report)

Jules Bertrand[1], Adam Moutonnet[2]

[1]jules.bertrand@berkeley.edu, [2]adam_moutonnet@berkeley.edu

I. Introduction

Atari games are widely known and have been played by millions of people worldwide. They were one of the first test cases for state-of-the-art reinforcement learning (RL) algorithms developed by top Machine Learning scientists from Google, Facebook, etc... Indeed, they embody the typical environment to serve as an RL problem, and the results can easily be grasped by anyone that has played video games before, bringing more public acknowledgement to such research.

In an RL problem, intelligent agents evolve following a discretized horizon in an environment within which they take intelligent actions at each step depending on the observations they make of it. Each of these actions has an influence on the environment, changing its state steps after steps, and consequently the observations agents make of it. We can assign a goal to every agent, a goal he will try to reach through a sequence of actions he will take. To ensure that the agent learns what action to take depending on the observation he has from the environment, we introduce the concept of rewards and penalties. Depending on how the action he takes changes the state of the environment, we give him a reward or a penalty. The reward means he is on the right track, the penalty means the contrary. Hence, the goal of the agent at each step is to take the action that maximizes the future expected 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 that good (it is a penalty). On the other hand, if they win it means that their actions were not that bad (it is a reward). That is why Atari Games are widely used to demonstrate the performances of reinforcement learning algorithms. Plus, the awesome Python library *Open AI* implements all of these games, making the Atari environment accessible for everyone.

Our goal throughout this project is to explore the two main reinforcement learning methods used nowadays that are *Q-Learning* [1] and *Policy Gradient* [2]. We will implement and test the performances of the original methods on Atari games provided by *Open AI* and bring to light the defaults and the advantages of both methods. We will then implement some of the newest version of these methods, with the newest features making them more stable and scalable and bring to light the improvements made on the games. Finally, we will keep the most powerful method and explore the possibility of using it, with only one neural network to play more than one game.

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 one 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 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 dynamic 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 priory, and we denote by τ such trajectory. This trajectory has a certain probability to happen under a given policy π, and we denote by $\pi(\tau)$ this probability. Let $\gamma \in [0, 1]$ be a discounting factor, we define the discounted reward $R(\tau)$ associated to a trajectory τ

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

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

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

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

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

Policy Gradient

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 the parameters of the probability distribution followed by u_k. For example, in an environment where the two available actions would be left or right (*Tetris* game for example), we could imagine that u_k will follow a Bernoulli distribution $B(p)$, and that the neural network will predict the parameter p based on x_k.

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

$$\theta^* = \arg\max_\theta J(\theta) \text{ with } J(\theta) = E_{\tau\sim\pi_\theta}\left[R(\tau)\right]$$

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

$$\begin{aligned} J(\theta) &= E_{\tau\sim\pi_\theta}\left[R(\tau)\right] \\ &= \sum_{\tau\in\mathfrak{T}} \pi_\theta(\tau) R(\tau) \end{aligned}$$

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

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

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

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

And the gradient $\nabla_\theta \log \pi_\theta(\tau)$:

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

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

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

$$
\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta} \left[\left(\sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) \right) \left(\sum_k \gamma^k r(x_k, u_k) \right) \right]
$$

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 it has no influence on the gradient. Let $k \in \mathbb{N}$,

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

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 $\sum_{j<k} \gamma^j r(x_j, u_j)$ whose gradient is null according to the last proof:

$$
\begin{aligned}
\nabla_\theta J(\theta) &= E_{\tau \sim \pi_\theta} \left[\sum_k \nabla_\theta \log \pi_\theta(u_k \mid x_k) \left(\sum_{j \geq k} \gamma^j r(x_j, u_j) \right) \right] \\
&= \sum_k E_{\substack{x_{0:k} \\ u_{0:k} \sim \pi_\theta}} \left[E_{\substack{x_{k+1:} \\ u_{k+1:} \sim \pi_\theta}} \left[\nabla_\theta \log \pi_\theta(u_k \mid x_k) \left(\sum_{j \geq k} \gamma^j r(x_j, u_j) \right) \;\middle|\; x_0, u_0, \ldots, x_k, u_k \right] \right] \\
&= \sum_k E_{\substack{x_{0:k} \\ u_{0:k} \sim \pi_\theta}} \left[\gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) E_{\substack{x_{k+1:} \\ u_{k+1:} \sim \pi_\theta}} \left[\sum_{j \geq k} \gamma^{j-k} r(x_j, u_j) \;\middle|\; x_0, u_0, \ldots, x_k, u_k \right] \right] \\
&= \sum_k E_{\substack{x_{0:k} \\ u_{0:k} \sim \pi_\theta}} \left[\gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) E_{\substack{x_{k+1:} \\ u_{k+1:} \sim \pi_\theta}} \left[\sum_{j \geq k} \gamma^{j-k} r(x_j, u_j) \;\middle|\; x_k, u_k \right] \right] \\
&= \sum_k E_{\substack{x_{0:k} \\ u_{0:k} \sim \pi_\theta}} \left[\gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k) \right]
\end{aligned}
$$

$$= E_{\tau\sim\pi_\theta}\left[\sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k)\right] \tag{1}$$

Passage from row 3 to row 4 is possible because we are dealing with 1^{st} order Markov Chains. Hence, we can derive a corresponding loss $J(\theta)$ by "integrating" this formula of the gradient:

$$J(\theta) = E_{\tau\sim\pi_\theta}\left[\sum_k \gamma^k \log \pi_\theta(u_k \mid x_k) Q^{\pi_\theta}(x_k, u_k)\right] \tag{2}$$

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

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

With $\tau^{(i)} = (x_0^{(i)}, u_0^{(i)}, x_1^{(i)}, u_1^{(i)}, \dots)$. Note that we got rid of γ^k because it is never considered in practice. 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 by

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

This method is called *REINFORCE* and is in fact not that much 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 1 tells us that it is equivalent to change the policy in a way that it increases the likelihood of "good" actions (actions that has a high Q-value for the current state).

One of the most known challenge with Policy Gradient is the high variance of the processed gradient. This can lead to brutal change in policy during the training, causing convergence issues. One solution to this problem developed in [3] 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 $u_k, x_{k+1}, u_{k+1}, \dots$ hence will not have any effect on the parameter update. However, this function is a priory not known, 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. We define the advantage function

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

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

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

This method is called *Advantages Actor Critic* (or A2C) and is much more used than REINFORCE thanks to its better convergence. The *Critic* is there 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

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

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

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

However, even with an optimal baseline like the value function, brutal changes in the policy may still happen. A solution developed in [4] 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 we can use a known distribution to estimate another distribution:

$$\begin{aligned}
\nabla_\theta J(\theta) &= E_{\tau\sim\pi_\theta}\left[\sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) A_\psi^{\pi_\theta}(x_k, u_k)\right] \\
&= \sum_{\tau\in\mathfrak{T}} \pi_\theta(\tau) \sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) A_\psi^{\pi_\theta}(x_k, u_k) \\
&= \sum_{\tau\in\mathfrak{T}} \frac{\pi_{\theta_{old}}(\tau)}{\pi_{\theta_{old}}(\tau)} \pi_\theta(\tau) \sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) A_\psi^{\pi_\theta}(x_k, u_k) \\
&= \sum_{\tau\in\mathfrak{T}} \pi_{\theta_{old}}(\tau) \frac{\pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)} \sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) A_\psi^{\pi_\theta}(x_k, u_k) \\
&= E_{\tau\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)} \sum_k \gamma^k \nabla_\theta \log \pi_\theta(u_k \mid x_k) A_\psi^{\pi_\theta}(x_k, u_k)\right] \\
&= E_{\tau\sim\pi_{\theta_{old}}}\left[\sum_k \gamma^k \frac{\pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)} \frac{\nabla_\theta \pi_\theta(u_k \mid x_k)}{\pi_\theta(u_k \mid x_k)} A_\psi^{\pi_\theta}(x_k, u_k)\right] \\
&= E_{\tau\sim\pi_{\theta_{old}}}\left[\sum_k \gamma^k \prod_j \frac{\pi_\theta(u_j \mid x_j)}{\pi_{\theta_{old}}(u_j \mid x_j)} \frac{\nabla_\theta \pi_\theta(u_k \mid x_k)}{\pi_\theta(u_k \mid x_k)} A_\psi^{\pi_\theta}(x_k, u_k)\right] \\
&= E_{\tau\sim\pi_{\theta_{old}}}\left[\sum_k \gamma^k \prod_{j\neq k} r_j(\theta) \frac{\nabla_\theta \pi_\theta(u_k \mid x_k)}{\pi_{\theta_{old}}(u_k \mid x_k)} A_\psi^{\pi_\theta}(x_k, u_k)\right]
\end{aligned}$$

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 get the following approximation:

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

And that's exactly what PPO is doing by considering the surrogate objective function

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

Deep Q-Learning

With Q-learning, we aim at predicting the Q-values associated to every possible state-action pairs (x_k, u_k) $Q^\pi(x_k, u_k)$, with π the best available policy. Hence, the best available action u_k^* to take at step k will simply be

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

These values are stored into a Q-table in which the rows are the states x_k and the columns are the actions u_k. At each step of the algorithm, given the learning rate α:

1) Choose an action u in the space of possible actions given the agent's current state x.
2) Take the action and observe the new state $x' = f(x, u)$, and the new space of possible actions $u' \in \mathcal{U}_{x'}$.
3) Compute the maximum future expected reward for this transition state + action $\rightarrow$ new state: $r(x, u) + \gamma \max_{u' \in \mathcal{U}_{x'}} Q^\pi(f(x, u), u')$.
4) Compute the error between the expected reward we actually have and the value stored in the Q-table for this transition: $r(x, u) + \gamma \max_{u' \in \mathcal{U}_{x'}} Q^\pi(f(x, u), u') - Q^\pi(x, u)$.
5) Update the Q-table:

$$Q^\pi(x, u) \leftarrow Q^\pi(x, u) + \alpha \left[r(x, u) + \gamma \max_{u' \in \mathcal{U}_{x'}} Q^\pi(f(x, u), u') - Q^\pi(x, u) \right]$$

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

This algorithm works very well when the states and actions spaces are small, but becomes inefficient as their sizes grow.The Q-table becomes to big to be fully updated in a reasonable time. Here, the solution we chose is the 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.

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

At each step, we observe a sequence of transitions. For each transition, the agent in state x takes an action u and goes to state $x' = f(x, u)$. Then we use the DNN to estimate $\hat{Q}(x', u', \theta) =$ for all possible u' and choose the action that yields the best possible future expected reward, and to estimate $\hat{Q}(x, u, \theta)$. The target Q-value has the same expression as in basic Q-learning: $r(x, u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x, u), u', \theta)$. 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 update directly the Q-values. Instead, it updates the weights (or parameters, θ) of the DNN. In order to do that, we will use the following loss function over the observation sequence [5]:

$$L_i(\theta_i) = \mathbb{E}_{x,u,r,x'} \left[\left(r(x, u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x, u), u', \theta_i) - \hat{Q}(x, u, \theta_i) \right)^2 \right] \tag{3}$$

Then the weights θ of the DQN are updated, given learning rate α:

$$\theta_{i+1} = \theta_i + \alpha \left(r(x,u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta_i) - \hat{Q}(x,u,\theta_i) \right) \nabla_{\theta_i} \hat{Q}(x,u,\theta_i) \tag{4}$$

However, as stated in [5], *"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"*. The main reasons are that Q-values are always modified, hence data distribution and policy always change, leading to unstability, the correlation between target estimations and the actual Q-value estimations as they are computed with the same DNN, and the correlation in the observation sequence distribution. To address these issues, [5] suggests two solutions:

- **Experience replay**: all transitions are stacked into a list, the *replay memory*, from which we will sample some transitions to train the DNN. It reduces correlation over the sequence of transitions and smoothes the data distribution.
- **Double Deep Q-network**: we make a copy a the DNN to compute the targets. This copy, named targetDNN for clarity, 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 $\hat{Q}(x,u,\theta_i)$. Thus targets Q-values will be calculated with older weights θ_i^-.

Thus when using experience replay, the sequence of observations used to train the DNN is sampled from the replay memory. When using Double Deep Q-learning, the loss function (3) and weights updates (4) must be fed with the following error:

$$r(x,u) + \gamma \max_{u' \in \mathcal{U}_{x'}} \hat{Q}(f(x,u), u', \theta_i^-) - \hat{Q}(x,u,\theta_i) \tag{5}$$

III. A First Implementation of the Methods

Before we bend over Atari games, which state space and action space are already a bit complicated, we start by implementing both methods on a very simple application game that is CartPole (see Figure 1).

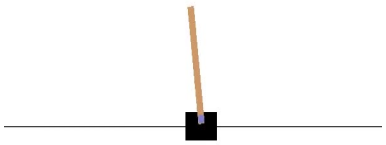

Fig. 1. Screenshot of the game CartPole-V0 from *Open AI*

The principle is really easy. A pole attached to a cart stands in equilibrium over this cart, and the purpose is to push the cart left or right to make the pole stay in equilibrium for the maximum number of steps. Here, the state space $\mathcal{X}$ is $\mathbb{R}^4$, and at step k the vector x_k summarizes the position and the speed of the cart, and the angle and the 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 dynamic and the solution of such a game could be easily derived, but we will let 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 describes the way we apply policy gradient to this problem.

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

Algorithm 1: Policy Gradient Method

main initialization: Choose an initial parameter θ_0, a learning rate α and a discount rate γ.
for $n = 0, \ldots$ **do**
  state initialization: Get x_0 from the game, initialize k to 0, the main trajectory τ to (), and the associated list of rewards ρ 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 τ
    Add (r_k) to ρ
    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

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 $(x_i, u_i, r_i, x_{i+1})_{0 \leq i \leq H-1}$ 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 targetDNN if Double Deep Q-learning is chosen) and updates the DNN weights θ by optimizing. Every p optimization steps, the targetDNN us updated. Algorithm 2 describes the way we apply Deep Q-learning to this problem.

Algorithm 2: Deep Q-learning

main initialization: Choose initial parameter θ_0, learning rate α, discount rate γ, size of the minibatches m, capacity of the replay memory C_L, sequence $(\epsilon_k)_{k\in\mathbb{N}}$, frequency of target network updates N_T, total number of optimization steps N_{opti}

for $n = 0, \dots$ **do**

  state initialization: Get x_0 from the game, initialize k to 0 and replay memory L_{replay} to $[\,]$

  while *True* **do**

    With probability ϵ_k select a random action u_k

    Otherwise select $u_k = \arg\max_{u\in\mathcal{U}_{x_k}} \hat{Q}(x_k, u, \theta_k)$

    $x_{k+1} \leftarrow f(x_k, u_k)$

    $r_k \leftarrow r(x_k, u_k)$

    Add (x_k, u_k, r_k, x_{k+1}) to L_{replay}

    if $N_{opti}\%N_T$ = *0* **then**

      Update target network: $\theta^- = \theta_k$

    end

    if $len(L_{replay}) > C_L$ **then**

      Sample random minibatch of m transitions (x_i, u_i, r_i, x_{i+1}) from L_{replay}

      Compute targets: $t_i = \begin{cases} r_i \text{ if } x_{i+1} = terminated \\ r_i + \max_{u'\in\mathcal{U}_{x'}} \hat{Q}(x_{i+1}, u', \theta^-) \text{ otherwise} \end{cases}$

      Compute Q-values $q_i = \hat{Q}(x_i, u_i, \theta_k)$

      Compute loss $L_k(\theta_k)$

      Perform gradient descent: $\theta_{k+1} \leftarrow \theta_k + \alpha\nabla_\theta L_k(\theta)|_{\theta=\theta_k}$

    end

    if x_{k+1}= *terminated* **then**

      Break the loop

    end

    $k \leftarrow k + 1$

    $x_k \leftarrow x_{k+1}$

  end

end

REFERENCES

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

[2] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[3] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, et al. Human-level control through deep reinforcement learning. *Nature 518, 529–533*, 2015.