ÉCOLE CENTRALE DE LYON

PROJET REPORT

Playing Pacman with Reinforcement Learning

Réalisé par :

Jaafar BEN ABERRAZAK

Mohamed Amine BOUAYAD

Mouhaned CHEBAANE

Issame SARROUKH

Khaled YAACOUBI

Soumaya ZAMMIT

Encadrant :

Alexandre SAIDI

Année :

2018 - 2019

Abstract

This report covers first the research study on Reinforcement Learning, a paradigm based on reward and punishment. Second, this report explains the different phases to design and the develop a reinforcement learning agent that is able to play the game of arcades MsPacman. This project is achieved as an option project at Ecole Centrale de Lyon.

Key words : Reinforcement Learning(RL), MDP, policy, agent, Deep Reinforcement Learning, MsPacman.

Résumé

Ce rapport couvre d'abord une étude bibliographique qui porte sur l'apprentissage par renforcement, un paradigme basé sur la récompense et la punition. Deuxièmement, ce rapport explique les différentes phases de la conception et le développement d'un agent d'apprentissage par renforcement qui est en mesure de jouer le jeu d'arcades MsPacman. Ce projet est réalisé comme un projet d'option à l'École Centrale de Lyon.

Key words : apprentissage par renforcement, MDP, stratégie, agent, Deep Reinforcement Learning, MsPacman.

Table des matières

Table des figures

Introduction

Reinforcement learning is born from the encounter between experimental psychology and computational neuroscience. It holds in some simple key concepts based on the fact that the intelligent agent :

- Observe the effects of his actions deduced from his observations the quality of his actions.
- Improve its future actions To define a framework for this process, we will teach him how to learn.

Man works this way when learning to bike, walk, etc. and scientists can train animals to do amazing things this way.

The "smart" agent decides to perform an action based on his state to interact with his environment. The environment (ie the user responsible for programming the agent) sends back a reinforcement in the form of a positive or negative reward. Then charge the agent to maximize this reinforcement. The reward here corresponds to the criterion to be optimized for the feedback loop in theory of optimal control. It is through this feedback loop that the agent improves.

Our work is presented throughout this report accordingly to the following plan. In the first section, we present our research study. We start by defining the Markov Decision Process as the classic tool to model a reinforcement learning problem. In addition, we provide a brief literature review on the different approaches of Reinforcement Learning.

In the second section we present our application. Throughout this section, we begin with a presentation of the context of our agent before shedding the light on the different technologies used to build the project some of the most important keywords of our work and provide a brief literature review and we justify our choice of algorithm to build our agent. Then we describe the development phases of our agent and present the results of each phase and present the main interface using screen-shots.

Finally, we conclude our work by giving future prospects and potential upgrades for our agent.

1 Markov Decision Process

To understand the MDP, first we have to understand the Markov property.
Markov property review :

— Given a sequence : $\{x_1, x_2, ..., x_t\}$
— Generally, this can't be simplified : p$\{x_t|x_{t-1}, x_{t-2}, ..., x_1\}$
— First-order Markov (or just "Markov") : p$\{x_t|x_{t-1}, x_{t-2}, ..., x_1\}$ = p$\{x_t|x_{t-1}\}$
— Second-order Markov : p$\{x_t|x_{t-1}, x_{t-2}, ..., x_1\}$ = p$\{x_t|x_{t-1}, x_{t-2}\}$

Any reinforcement learning task with a set of states, actions, and rewards, that follows the Markov property is a Markov Decision Process. Formally speaking, a Markov Decision Process is a five-tuple made up of the set of **states**, set of **rewards**, set of **actions**, the **state-transition** probabilities, and the **reward** probabilities and the **discount factor**

Markov Decision Processes (MDPs)
— We've been looking at MDPs, just not calling them by name.
— Any RL task with a set of states, actions, and rewards, that follows the Markov property, is a MDP.
— MDP is defined as the collection of :
  — Set of states
  — Set of actions
  — Set of rewards
  — State-transition probabilities, reward probabilities (as defined jointly earlier)
  — Discount factor
— Often written as a 5-tuple

Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows).

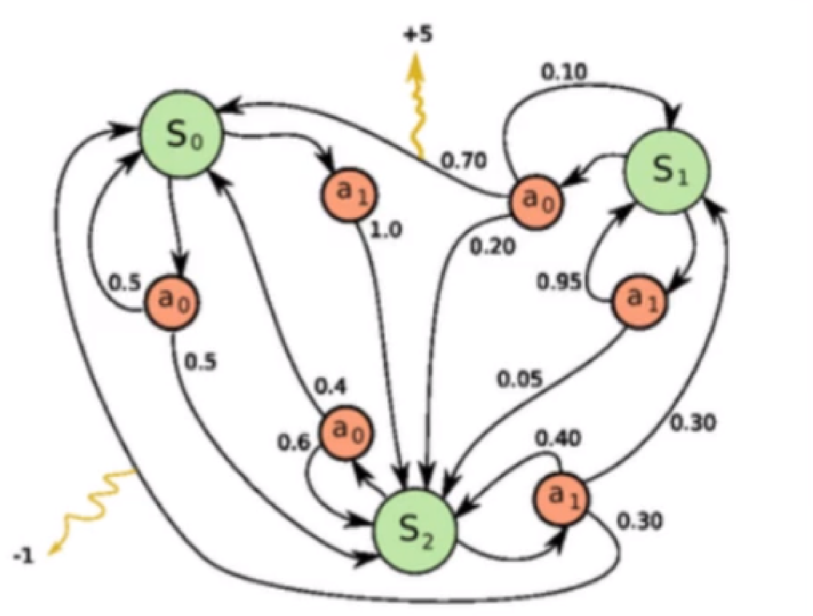


FIGURE 1 – state-transition diagram

The way we make decisions for what actions to do in what states is called a policy. We usually denote a policy with the symbol pi. Technically, the policy is not part of the MDP itself, but it's part of the solution .

Policy :

— One more piece to complete the puzzle - the policy (denoted by π)

— Technically π is not part of the MDP itself, but it, along with the value function, form the solution.

— There's no "equation" for it

— How do you write epsilon-greedy as an equation ? It's more like an algorithm

— The only exception is the optimal policy, which can be defined in terms of the value function.

— Think of π as shorthand for the algorithm the agent is using to navigate the environment.

We are interested in is measuring the total future reward.
So everything from T plus one and onward. We call this the return and we use the symbol G to represent it.
Notice how it doesn't depend on the reward at the current time. This is because, strictly speaking, when I arrive at a state, I receive the reward for that state.
There is nothing I can predict about it since it has already happened.

Now think of a very long task, a task containing a sequence that is thousands of steps long.
Our goal is to maximise your total reward.

Total reward :

— We are interested in measuring total future reward
— Everything from $t + 1$ onward
— We call this the return, G(t)
— Does not count current reward R(t)

$$\mathbf{G(t)} = \sum_{\tau=1}^{\infty} T(t+\tau)$$

But is there a difference between getting a reward now and getting that same reward 10 years from now ?
And so we introduce what is called a discount factor on the future rewards.
γ is a discount factor, where γ in [0, 1] .
It informs the agent of how much it should care about rewards now to rewards in the future.
If (γ= 0), that means the agent is short-sighted, in other words, it only cares about the first reward.
If (γ= 1), that means the agent is far-sighted, i.e. it cares about all future rewards.
What we care about is the total rewards that we're going to get.

Discount factor :

— Gamma = 1 : don't care how far in the future reward is, weigh all equally
— Gamma = 0 : truly greedy, only try to maximize immediate reward
— Usually we choose something close to 1, i.e. 0.9
— short episodic task ; maybe don't discount at all
— "The further you look into the future, the harder it is to predict"

$$\mathbf{G(t)} = \sum_{\tau=0}^{\infty} \gamma^{\tau} R(t+\tau+1)$$

2 Model-Based Reinforcement Learning

Reinforcement learning has led to the development of highly efficient neural network learning algorithms. They are in particular used in a robotic frame, for the manipulation of objects by robotic arms. An effective architecture in this context is the Modular Actor-Critical architecture, where an actor realizes the learned policy and the critic uses the feedbacks to update the value of the states as well as the parameters of the policy. This approach is called 'model-free' because learning can refine the behavior without having an explicit model of the environment. This can, however, lead to some disadvantages, especially in terms of learning time which can become very long because many examples are necessary for training. This disadvantage can become prohibitive when the learning space is very large or when the areas of space that are rewarded are few and sparse. In this case, learning without a model becomes impossible and other solutions must be found. The model-based approach has long been known to be a solution to this problem because, by explicitly using transition rules describing the dynamics of the world, it avoids spending a long time learning it. But it supposes that this model of the world is available,

In reinforcement learning, we have some state space $\mathcal{S}$ and action space $\mathcal{A}$ If at time t we are in state $s_t \in \mathcal{S}$ and take action $a_t \in \mathcal{A}$, we transition to a new state $s_{t+1} = f(s_t, a_t)$ according to a dynamics model $f: \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. The goal is to maximize rewards summed over the visited state : $\sum_{t=1}^{T-1} r(s_t, a_t, s_{t+1})$. Model-based RL algorithms assume you are given (or learn) the dynamics model f. Given this dynamics model, there are a variety of model-based algorithms. For this post, we consider methods that perform the following optimization to choose a sequence of actions and states to maximize rewards :

$$max_{a_1,\dots,a_{T-1},s1,\dots,s_T} r(s_t, a_t, s_{t+1}) \quad subject \quad to \quad f(s_t, a_t) = s_{t+1}$$

The optimization says to choose a sequence of states and actions that you maximize the rewards, while ensuring that the trajectory is feasible. Here, feasible means that each state-action-next-state transition is valid. For example, in the image below if you start in state s_t and take action a_t , only the top s_{t+1} results in a feasible transition.

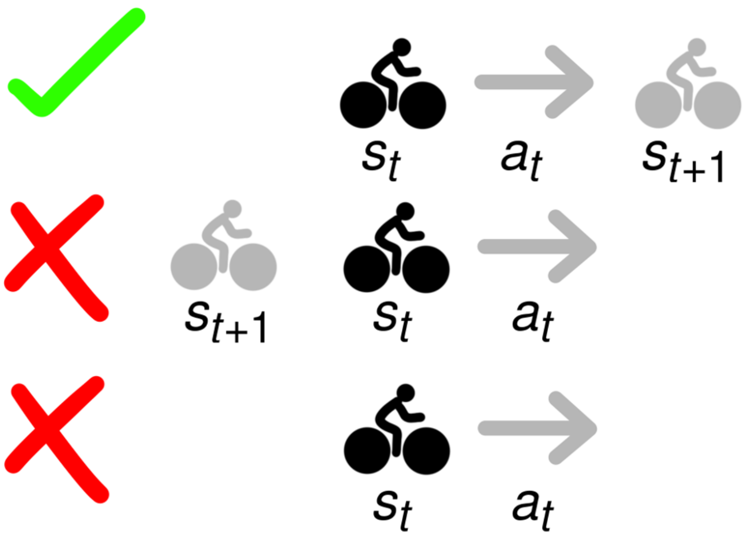


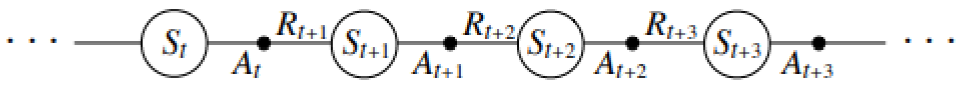
FIGURE 2 – Feasable actions

3 Value Learning

3.1 SARSA :On-Policy TD Control

In this section, we will turn to the use of TD prediction methods for the control problem. To do so, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. we face the need to trade off exploration and exploitation, and approaches fall into two main classes : on-policy and off-policy. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s,a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π. Recall that an episode consists of an alternating sequence of states and state–action pairs :



In this section, we are considering transitions from state–action pair to state–action pair, and learn the value of state–action pairs. Formally these cases are identical : they are both Markov chains with a reward process.

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
   Repeat (for each step of episode):
      Take action $A$, observe $R$, $S'$
      Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
      $S \leftarrow S'$; $A \leftarrow A'$;
   until $S$ is terminal
```

FIGURE 3 – Sarsa : An on-policy TD control algorithm

The figure above shows the SARSA : on policy TD control algorithm. The main goal of the algorithm is to solve the action-value function shown beyond and get to its global optimum for each iteration. Thus, an robust algorithm is a one that could converge to the optimum rapidly and get an the biggest reward at the end of the iteration process.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then Q(S_{t+1}, A_{t+1}) is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name Sarsa for the algorithm.
It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. The general form of the Sarsa control algorithm is given in Figure 9.

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on q. For example, one could use ϵ-greedy or ϵ-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ϵ-greedy policies by setting $\epsilon =$ 1/t), but this result has not yet been published in the literature.

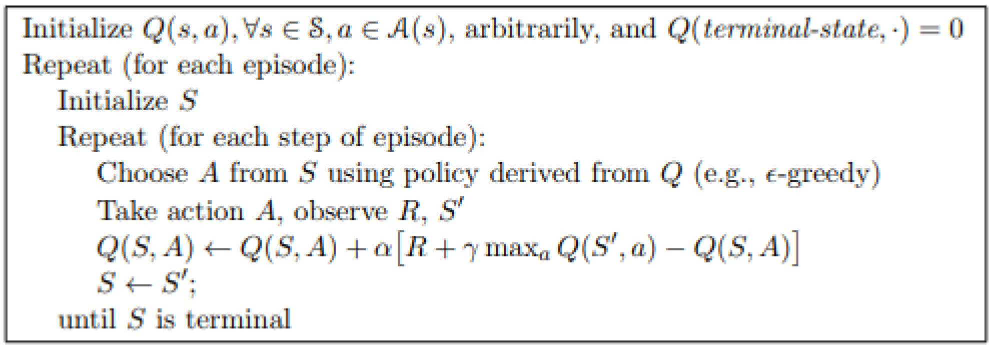
3.2 Q-Learning

3.2.1 Q-Learning : Off-Policy TD Control

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning.Its simplest form, one-step Q-learning, is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. This is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q. The Q-learning algorithm is shown in procedural form in Figure



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

FIGURE 4 – Q-learning : An off-policy TD control algorithm

The rule shown in the figure 10 updates a state–action pair, so the top node, the root of the backup, must be a small, filled action node. The backup is also from action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these "next action" nodes with an arc across them.

3.3 DQN : Deep Q-Networks

In this section we will walk through the DQN algorithm. Although Q-learning is a very powerful algorithm, its main weakness is lack of generality. If you view Q-learning as updating numbers in a two-dimensional array (Action Space * State Space), it, in fact, resembles dynamic programming. This indicates that for states that the Q-learning agent has not seen before, it has no clue which action to take. In other words, Q-learning agent does not have the ability to estimate value for unseen states. To deal with this problem, DQN get rid of the two-dimensional array by introducing Neural Network.
DQN leverages a Neural Network to estimate the Q-value function. The input for the network is the current, while the output is the corresponding Q-value for each of the action.

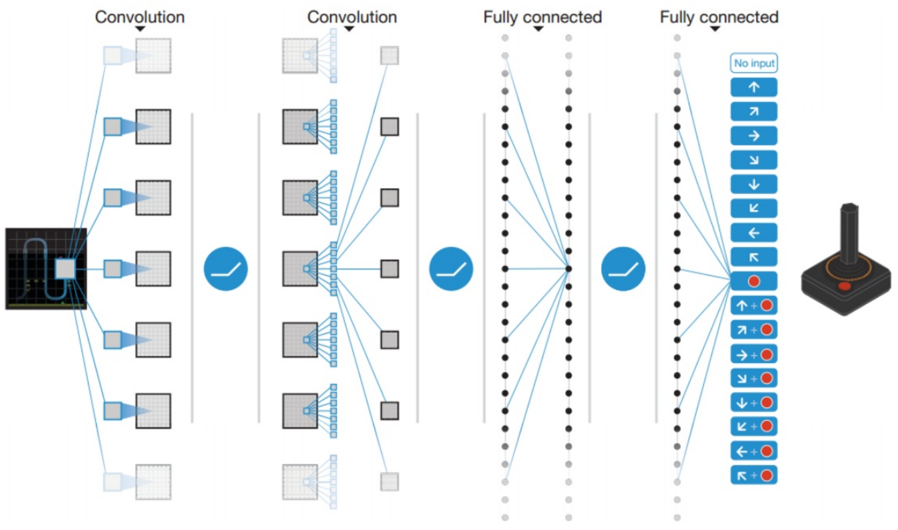


FIGURE 5 – DQN architecture

In 2013, DeepMind applied DQN to Atari game, as illustrated in the above figure. The input is the raw image of the current game situation. It went through several layers including convolutional layer as well as fully connected layer. The output is the Q-value for each of the actions that the agent can take.
The question boils down to : **How do we train the network ?**
The answer is that we train the network based on the Q-learning update equation. Recall that the target Q-value for Q-learning is :

$$r_j + \gamma max_{a'} Q_\theta(s_{j+1}, a')$$

The loss function for the network is defined as the Squared Error between target Q-value and the Q-value output from the network.

3.4 Double DQN

The difference between Double DQN and the regular DQN lies in the fact that the second one often overestimates the Q values of the potential actions to be undertaken in a given state. The agent has a trouble choosing the best policy, if the Q-value chosen is not optimal for the given action. To remedy this, a trick is proposed were the max over Q-values is used to calculate the target Q-value. During the training phase, the primary network choose the action, the target network generates the target Q-value generation. Relying on the trick explained above, the training is much faster, the result is more reliable, and the overestimation is greatly reduced. The new equation for updating the target value is written below :

$$Q_{Target} = r + \gamma Q(s', argmax(Q(s', a, \theta), \theta'))$$

3.5 Dueling DQN

The dueling DQN is based in the computation of the two functions : The advantage and value functions, this model combines them together at the final layer into a single Q-function.

First, it is necessary to explain the two function, as indicated previously, the Q-values are the central components to evaluate the quality of an action in a certain state. The action given state written in the form of $Q(s, a)$ can be separated into two functions, the first is the value function $V(s)$ which indicates the quality of a given state, the second is the advantage function $A(a)$, which indicates how much an action is better than others. This combination of these two function can be written formally as follows :

$$Q(s, a) = V(s) + A(a)$$

The neural network is divided into two streams sharing a convolution base, as shown below :

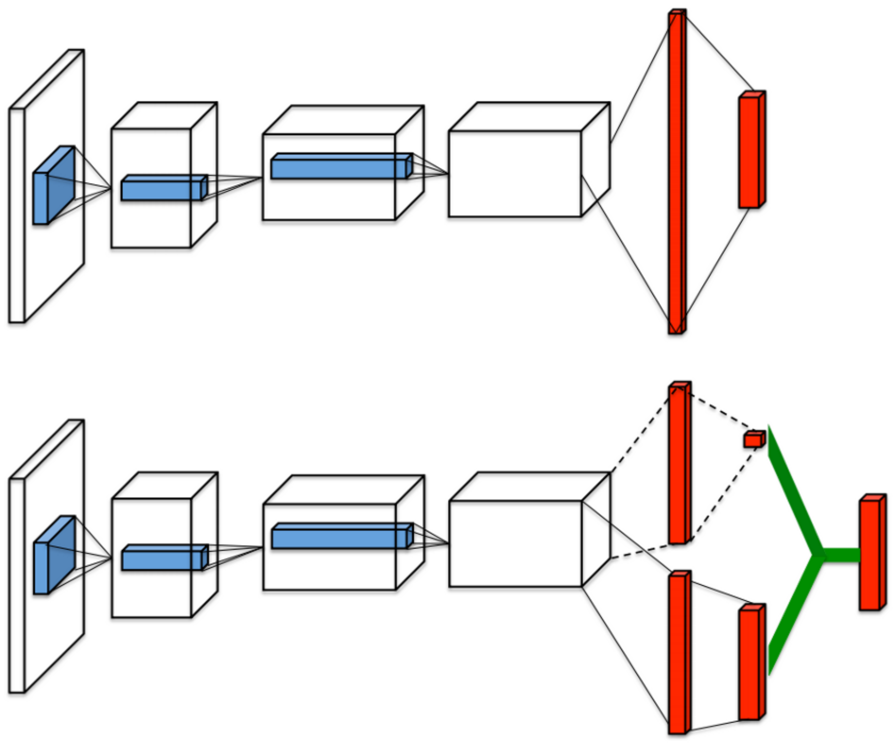


FIGURE 6 – Vanilla DQN architecture (top) vs. Dueling DQN (bottom)

This seems counterintuitive, but it works and allows a better estimate than the normal DQN.

3.5.1 Asynchronous n-step Q-learning

The algorithm is somewhat unusual because it operates in the forward view by explicitly computing n-step returns, as opposed to the more common backward view used by techniques like eligibility traces (Sutton & Barto, 1998).
We found that using the forward view is easier when training neural networks with momentum-based methods and backpropagation through time. In order to compute a single update, the algorithm first selects actions using its exploration policy for up to t_{max} steps or until a terminal state is reached. This process results in the agent receiving up to t_{max} rewards from the environment since its last update.
The algorithm then computes gradients for n-step Q-learning updates for each of the state-action pairs encountered since the last update. Each n-step update uses the longest possible n-step return resulting in a one-step update for the last state, a two-step update for the second last state, and so on for a total of up to t_{max} updates. The accumulated updates are applied in a single gradient step.

4 Policy-based methods

In the last section, we talked about value-based reinforcement learning algorithms. In order to choose which action to pick given a state St, we select the action with the highest value-related function. As a consequence, in value-based learning, a policy exists only because of these action-value estimates.
Another natural approach in reinforcement learning is to directly parameterize the policy distribution instead of value function. Thus, in this section, we'll learn about policy-based reinforcement learning technique called Policy Gradients.

4.1 Introduction to policy-based methods

The policy-based methods are another different approach in reinforcement learning. Instead of learning a value function that tells us what is the expected sum of rewards given a state and an action such as in the value-based method Q-Learning and Sarsa, we learn directly the policy function that maps state to action. This way we skip the observation and estimation phases.

Ones of the most immediate advantages of this approach are :

— A better convergence property : these type of methods tend to be more stable so we find ourselves smoothly updating the policy and avoiding the the big oscillation while training with value-based methods because we would be just making infinitesimal incremental changes in our policy which allows us to move toward the direction that makes the policy better. So because we follow the gradient to find the best parameters, we're guaranteed to converge on a local maximum (worst case) or global maximum (best case).

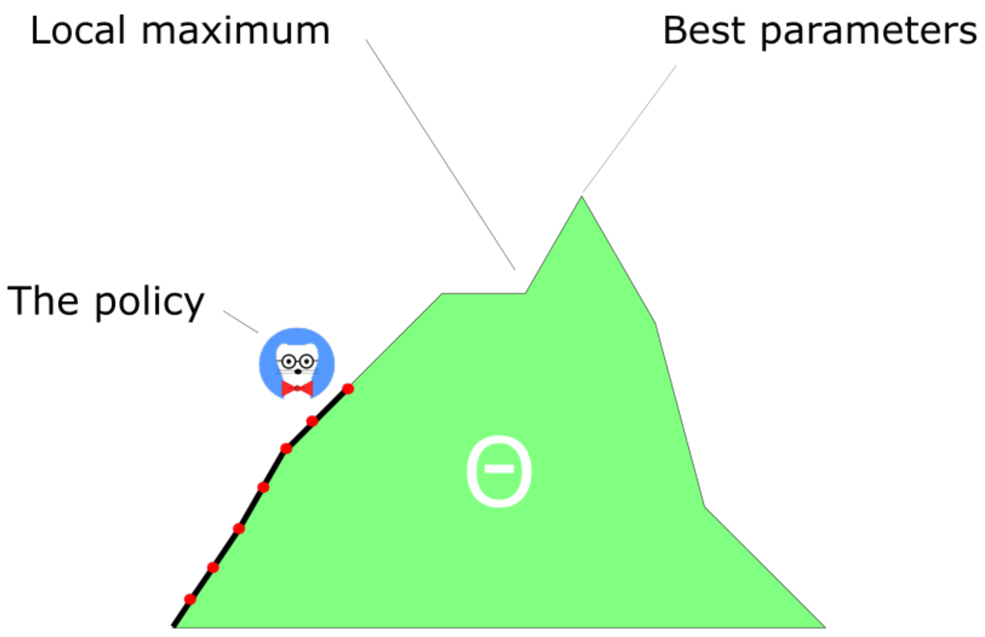


FIGURE 7 – Convergence of policy-based methods

— These methods are effective in high dimensional and continuous action spaces. Unlike the value based methods where we have to work out how to compute the maximum expected future reward for each possible action, at each time step, given the current stat, in policy-based methods we're just going to adjust the parameters of our policy directly so we never have to solve this maximization problem.

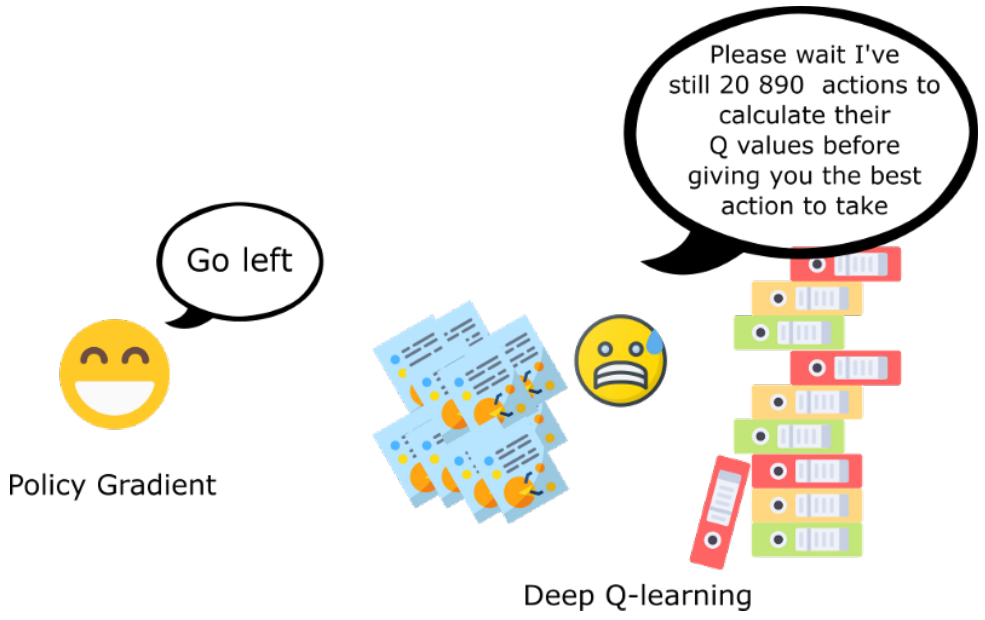

Please wait I've still 20 890 actions to calculate their Q values before giving you the best action to take
Go left
Policy Gradient
Deep Q-learning

FIGURE 8 – Effectiveness of policy-based methods in high dimensional

— Policy-based methods can learn stochastic policies and not only deterministic ones while value functions can't.

However, these methods can present some drawbacks. The first negative point include the fact that policy-based methods typically converge to a local rather than global optimum. The second negative point is that evaluating a policy is typically inefficient and introduce a high variance problem.

4.2 Policy Search

4.2.1 Policy Objective Functions

Given the policy π_θ(s, a) with parameters θ, our goal consist of finding the best θ. But how can we measure the quality of a policy π_θ ? To measure how good our policy is, we

use a function called the objective function (or Policy Score Function) that calculates the expected reward of policy.

There are three main objective functions that allows us ti know how good is our policy :

1. **The start value :**
   In the case of an episodic environments we can use the start value : $J_1(\theta) = V^{\pi_\theta}(s1) = E_{\pi_\theta}(v1)$. This function basically calculates the total reward we are going to get if we have a distribution over start state s1.

2. **The average value :**
   In the case of a continuing environments where we can't rely on a specific start state, we can use the average value : $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$ Where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for π_θ. In other words, in an environment that just goes on forever, there might not be a start state, so we consider the probability of ending up in any state under the probability of our policy π_θ multiplied by the value of that state on-wards so it's like averaging over the values of all states.
3. **The average reward per time-step :** $J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a$
   This function is similar to the previous one, but what really matters in this case is the most reward per time-step and not the average value.

4.2.2 Policy Optimisation

The Policy based reinforcement learning is an optimization problem aiming to find θ that maximizes $J(\theta)$ the policy objective method.

To solve this optimization problem, there are different approaches that do not use gradient such as :

— Hill climbing
— Simplex / amoeba / Nelder Mead
— Genetic algorithms

However, the experience has shown that a greater efficiency is often granted using gradient-based methods like :

— Gradient ascent/descent
— Conjugate gradient
— Quasi-newton

In this paper, we are going to focus basically on methods using gradient ascent and methods that exploit sequential structure.

4.2.3 Policy gradient ascent

In this paragraph, we are going to focus on maximizing the score function $J(\theta)$ by applying the gradient ascent on policy parameters.
The main idea about gradient ascent is to find the gradient to the current policy π that updates the parameters in the direction of the greatest increase, and iterate.
The figure following figure explains briefly the pseudo code of the gradient ascent algorithm.

$$
\begin{aligned}
&Policy : \ \pi_\theta \\
&Objective\ function\ : J(\theta) \\
&Gradient : \ \nabla_\theta J(\theta) \\
&Update : \theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)
\end{aligned}
$$

FIGURE 9 – The pseudo code of the gradient ascent algorithm

4.2.3.1 Computing Gradients By Finite Differences

We can evaluate the policy gradient of $\pi_\theta(s, a)$ using the Finite Differences method. For each dimension k in [1, n] we begin first by estimating the k_{th} partial derivative of objective function with respect to θ. Then, by perturbing θ by a small amount ϵ in k_{th} dimension

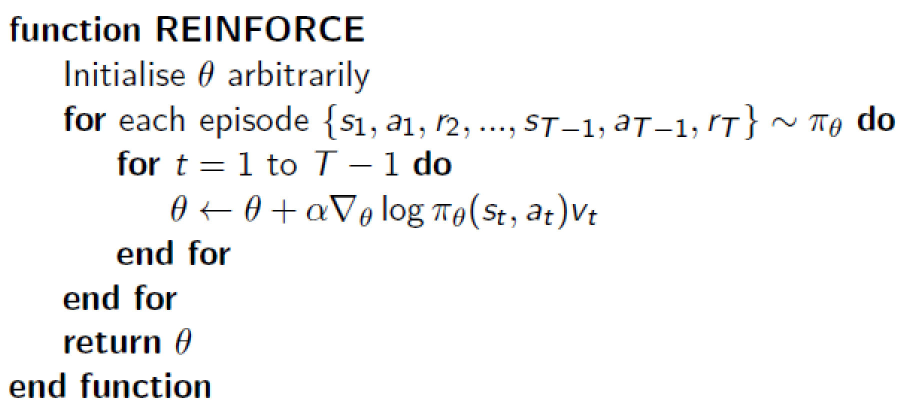
$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

where u_k is unit vector with 1 in k_{th} component, 0 elsewhere.

This method uses n evaluations to compute policy gradient in n dimensions. Despite its sleeplessness, this method is actually noisy and inefficient but sometimes it can be effective especially in the case of arbitrary policies, even if the policy is not differentiable.

4.2.4 Monte Carlo Policy Gradients

We compute now the policy gradient analytically using the Monte Carlo Policy Gradients.
The Monte Carlo Policy Gradients algorithm also called the REINFORCE algorithm which uses Monte Carlo rollout to compute the rewards. i.e. play out the whole episode to compute the total rewards.



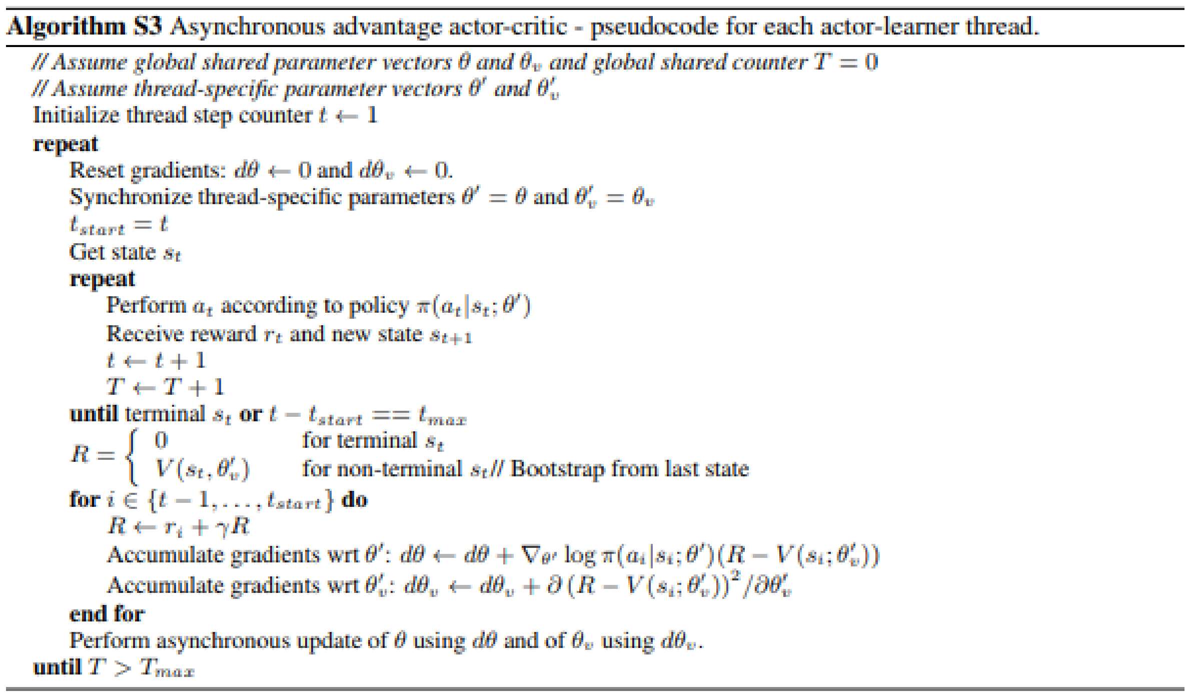
```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

]

Note that this algorithm works in the case of a continuous agent maximizing the averaged value. In the episodic case we better add the discount factor γ^t we talked about earlier in this paper where we maximize the value of start state instead. The updated value of in this case should be looking like this : $\theta = \theta + \alpha\gamma^t G_t \nabla_\theta log(\pi(A_t|S_t, \theta)$

4.3 Asynchronous advantage actor-critic(A3C)

The algorithm, which we call asynchronous advantage actor-critic (A3C),maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function V(s_t ; θ_v).
Like our variant of n-step Q-learning, our variant of actor-critic also operates in the forward view and uses the same mix of n-step returns to update both the policy and the value-function. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached. The pseudocode for the algorithm is presented below.



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

]

Figure 10 – The pseudo code of the A3C algorithm

As with the value-based methods we rely on parallel actorlearners and accumulated updates for improving training stability. Note that while the parameters θ of the policy and $_v$ of the value function are shown as being separate for generality, we always share some of the parameters in practice. We typically use a convolutional neural network that has one softmax output for the policy $\pi(a_t|s_t;\theta)$ and one linear output for the value function V(s_t ; θ_v), with all non-output layers shared.

We also found that adding the entropy of the policy π to the objective function improved exploration by discouraging premature convergence to suboptimal deterministic policies. This technique was originally proposed by (Williams & Peng, 1991), who found that it was particularly helpful on tasks requiring hierarchical behavior.

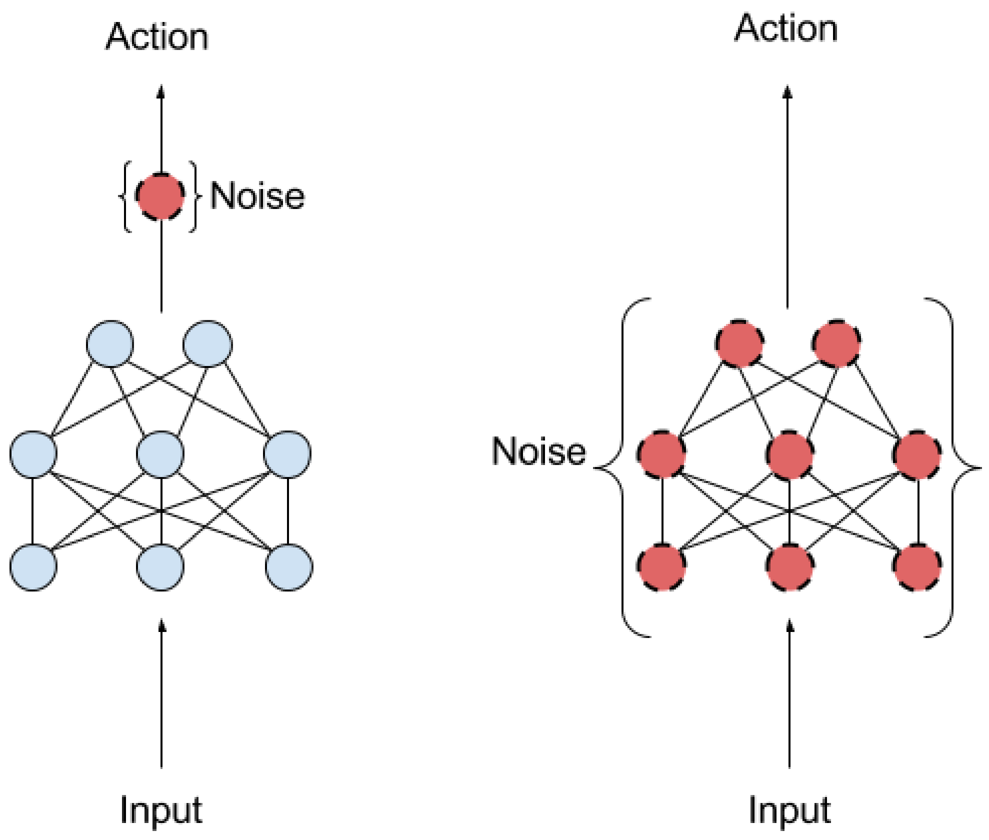
5 Exploration vs Exploitation

In this section we're going to focus on the obstacles that we face, in particular the exploration dilemma. The agent tend to stick to certain pattern of actions and because of that it usually don't take advantage of other paths and that is due to the lack of information because the agent doesn't know the expected reward. One of the solutions that we can use to solve this problem is using the noisy networks .

5.1 Noisy Networks for Exploration

How do we know how much exploration we'll need to solve a game like Ms. Pac-man ? The only real way to find out is to test a whole bunch of different values, a time consuming and expensive process. It would be much better if the agent could learn to control its exploration the same way it learns to control the rest of its decision-making—through gradient descent. In 2017, Fortunato et al. released the solution that made that possible : Noisy Networks.

Because we are calculating the Q-values as normal, but randomizing (adding noise) to our action selection, the epsilon Q greedy approach falls under the umbrella of action space noise exploration methods. In a Noisy Network, we instead inject noise into the weights of the neural network itself (the parameter space). As a result, the agent will consistently recommend actions that it didn't 'intend' to, ensuring exploration, and we can make action decisions based solely on the highest Q value output.



]

FIGURE 11 – The difference between action and parameter space noise

Specifically, Noisy Networks replace the Dense classifiers in the model with Noisy Dense layers, defined by the operation :

$$y = (\mu^{\omega} + \sigma^{\omega} \bigodot \varepsilon^{\omega})x + \mu^{\beta} + \sigma^{\beta} \bigodot \varepsilon^{\beta}$$

Where all of the parameters are learnable except for the epsilon terms, which are generated using factorized Gaussian noise before each training and inference step. The agent can then learn to manipulate the other variables, particularly the two sigmas, to tune out the noise but it can do so at the pace its environment demands, without any arbitrary interference from us.

Because the noise injection is element-wise multiplied by this term, the agent can learn to ignore the noise by setting sigma to 0. It's interesting to note that this method tends to naturally bottom-out above 0 just like we designed our epsilon schedule to ensuring at least some level of exploration for the duration of training.

6 Technical Choices

In this section, we discuss the technical choices we made to conceive and develop our Pacman agent. We start by presenting the language used in the development of the project followed by the library that helped us to simulate the environment for our agent. Afterwards, we defend our choice for the algorithms we used.

6.1 Development Language

The language we used throughout the development phase was mainly Python.
Python is an interpreted, high-level, general-purpose and more importantly an open source programming language. Python was very useful for the quick prototyping of our agent especially for preliminary analysis to see if the algorithm performs well. One other reason that makes us use Python for this project is its strong library support. In fact, Python comes with a number of very attractive libraries which can be applied out of the box. Through out the development phase we used :

— **Numpy :** it is a fundamental package for scientific computing with Python and we used it for a powerful N-dimensional array object and for the sophisticated (broadcasting) functions.

— **Tensorflow :** it's a free and open-source software library for dataflow and differentiable programming across a range of tasks. We used Tensorflow in our project to implement deep learning for our Deep D-Network algorithm.

— **Keras :** which is as well an open-source neural-network library written in Python, capable of running on top of TensorFlow to enable fast experimentation with deep neural networks.

— **Keras-rl :** This is a library that implements some state-of-the art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library Keras. This library helped us a lot to build a more sophisticated agent with Deep Q Learning (DQN).

6.2 Development Library

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents and gives us huge number of test environments to work on our RL agent's algorithms with shared interfaces for writing general algorithms and testing them.
Using Gym library, we were able to maximize the score in the Atari 2600 game MsPacman.

In this environment, the observation is the RAM of the Atari machine, consisting of (only !) 128 bytes. Each action is repeatedly performed for a duration of 4 frames.

There are two basic concepts in reinforcement learning : the environment (namely, the outside world) and the agent (namely, the algorithm you are writing). The agent sends actions to the environment, and the environment replies with observations and rewards (that is, a score).

The Gym interface library provides us with a unified environment interface. There is no interface for agents ; that part is left to us to develop. So basically, Gym take care of three main functions :

— Reset the environment's state and returns the observation.
— Step the environment by one timestep. Returns the observation, reward, done, info.
— Render one frame of the environment in a human friendly way. In the context of our application, it is going to pop up a window that shows us the game simulation.

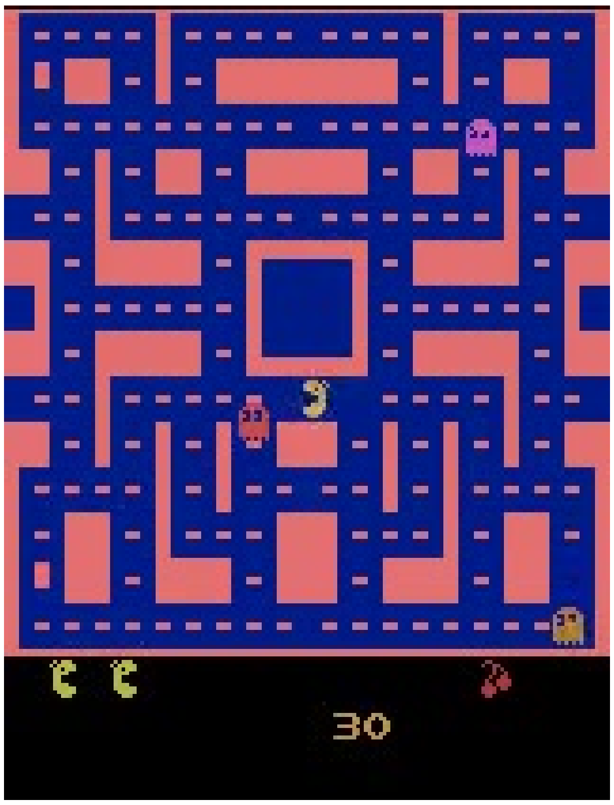


FIGURE 12 – Screenshot from MsPacman environment using Gym library

Thankfully Gym is also TensorFlow compatible which allowed us to use both libraries easilyf while implementing the Deep Q-Network algorithm.

6.3 The choice of the reinforcement learning algorithm

The algorithm we used for the implementation of our agent is mainly Deep Q-Network algorithm due to different reasons.

First of all, DQN has shown a great success with Atari games such as Space Invaders and Breakout due to its efficiency that surpassed the human capacities in these sort of games.

In addition to its effectiveness, DQN is simple to implement and understand and it facilitates the modeling of the environment. All we have to do is to feed the algorithm with the screen-shots of the game.

Last but not least, recent breakthroughs in computer vision have relied on efficiently training deep neural networks on very large training sets. The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than handcrafted features. These successes motivate our approach to reinforcement learning. By connecting a reinforcement learning algorithm to a deep neural network which operates directly on RGB images we can guarantee the best performance for our agent.

7 Solving Ms Pacman with reinforcement learning

Pac-Man, an emblematic character in the history of video games, is a yellow round character with a mouth. He must eat pac-gums and bonuses (in the form of fruit) in a labyrinth haunted by four ghosts. Four special gum pacs make ghosts vulnerable for a short period of time during which Pac-Man can eat them. The ghosts then turn blue and display an expression of fear signalled by small eyes and a broken line mouth and when a ghost is eaten, his eyes return to the central room of the labyrinth to make it normal again.

7.1 System Architecture

The architerure used to solve Ms Pacman consists of different components. The Pac-man implementation forms the environment and the DQN algorithm implemented in Python and representing the RL agent. Pacman-playing DQN process the screen using a convolutional neural network. Having limited computing ressources, we decided to pre-processed the game screen according to DeepMind's 2015 paper. we pre-process the raw input by reducing the original size to a more manageable 84 x 84 format and converting it to grayscale.

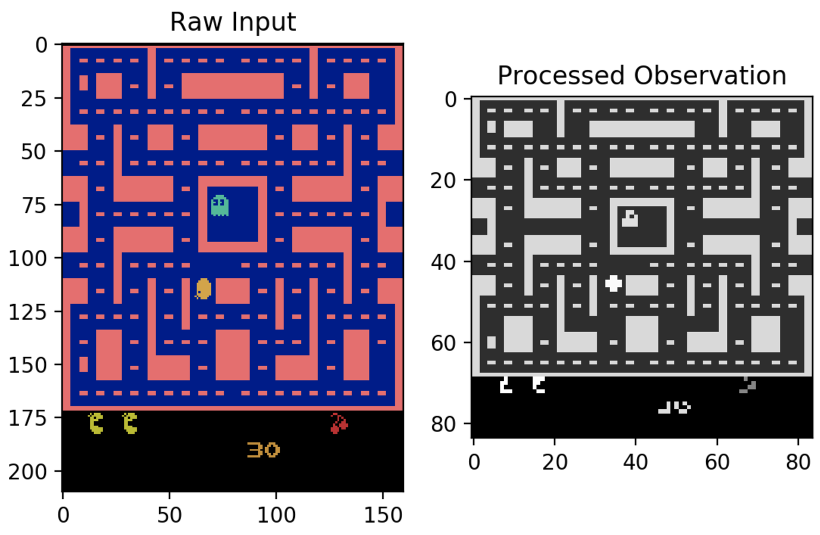


FIGURE 13 – Raw Input and preprocessed input

This preprocessing throws away color signals that may be helpful. For exemple, the network can't use the changing color of ghosts to know if they are vulnerable (but it still uses the fear signals). As mentioned above, the Markov property defines that state transitions depend only on the current state and that the previous history is not relevant. Applied to the Pac-man this means that no memory is required to store all the past images of the game. All ghosts in the Pac-man game moves in a certain direction, but without acceleration. to capture both position and direction information, a state is defined as the last 4 images of the game.

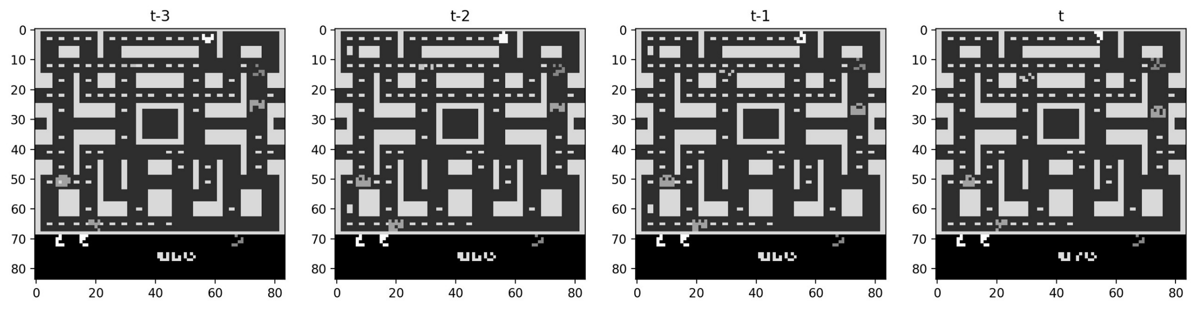


FIGURE 14 – 4 images included in the game state at t.

7.2 Results

In this section, we are going across the algorithms used for solving the game. Let's start by the first Algorithm.

7.2.1 Simple DQN : Deep Q Network

Something to know about DQNs (and RL algorithms in general) is that they are sample inefficient and expensive to train. The standard approach in the DQN literature is to run 200 million frame training sessions. That's about 930 hours of human-speed gameplay or roughly 20 days of training on a single P4000 GPU (at least for the final DQN variant we'll be getting to). We don't have the resources to pull something like that off for every version of the algorithm. Instead, we chose to run these comparison experiments for 10 million frames. An effective way to monitor and analyze an agent's success during training is to chart its cumulative reward at the end of each episode.

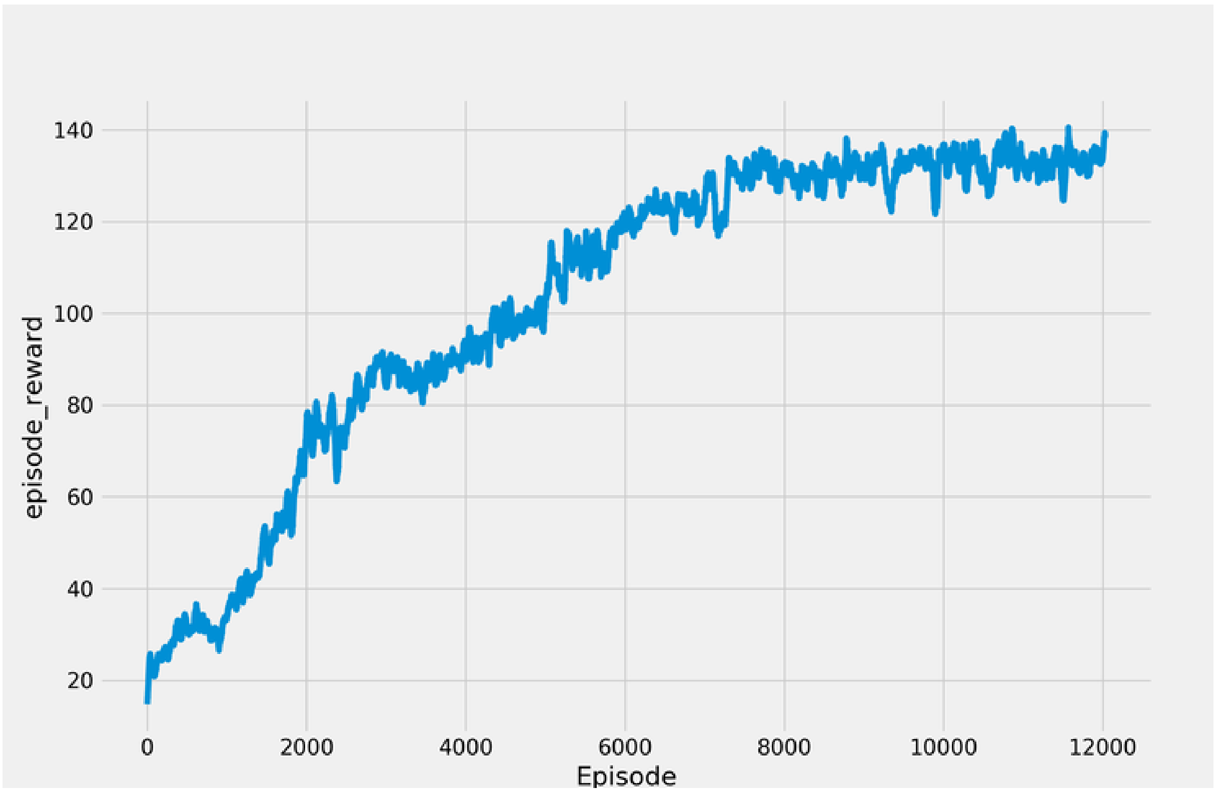


FIGURE 15 – The reward function by episode

The reward function is proportional to (but not equal to, and actually much less than) the in-game score, so it doesn't mean all that much on its own. However, the agent is clearly able to learn something about the game in order to consistently improve over time. If we look to the demo attached with this report, we will note that the agent has learned

to navigate around the maze. It even does a pretty good job of clearing large areas when there are still dense pockets to be collected. However, it has a very hard time finding its way back to the lonely few dots that slip away, and it seems to have total meltdowns whenever it finds itself stuck in the middle of two pockets and has to decide which one to go after first. It also does not seem to actively hunt down the ghosts for points when they become vulnerable, and only triggers that behavior by accident, on its way to gathering up all the dots in each corner of the maze.
Now let's move on to the next algorithm. In fact, it is the same one : DQN, but an improved version with some changes, this is the double dueling DQN, explained above.

7.2.2 Noisy Double Dueling Q-Learning

This algorithms use two improvements of DQN and a noisy neural network to deal with the exploitation problem.
The agent still facing the problem of the isolated points but it tries to catch the vulnerable ghosts as fast as possible in order to get better score. But the problem of that strategy is that the ghosts become more aggressive after being caught by the agent.

7.2.3 N-step Noisy Double Dueling Q-Learning

This is the final algorithm. Another DQN but using TD N-step rather than TD 1-step.
Here is the learning curve for our final algorithm :

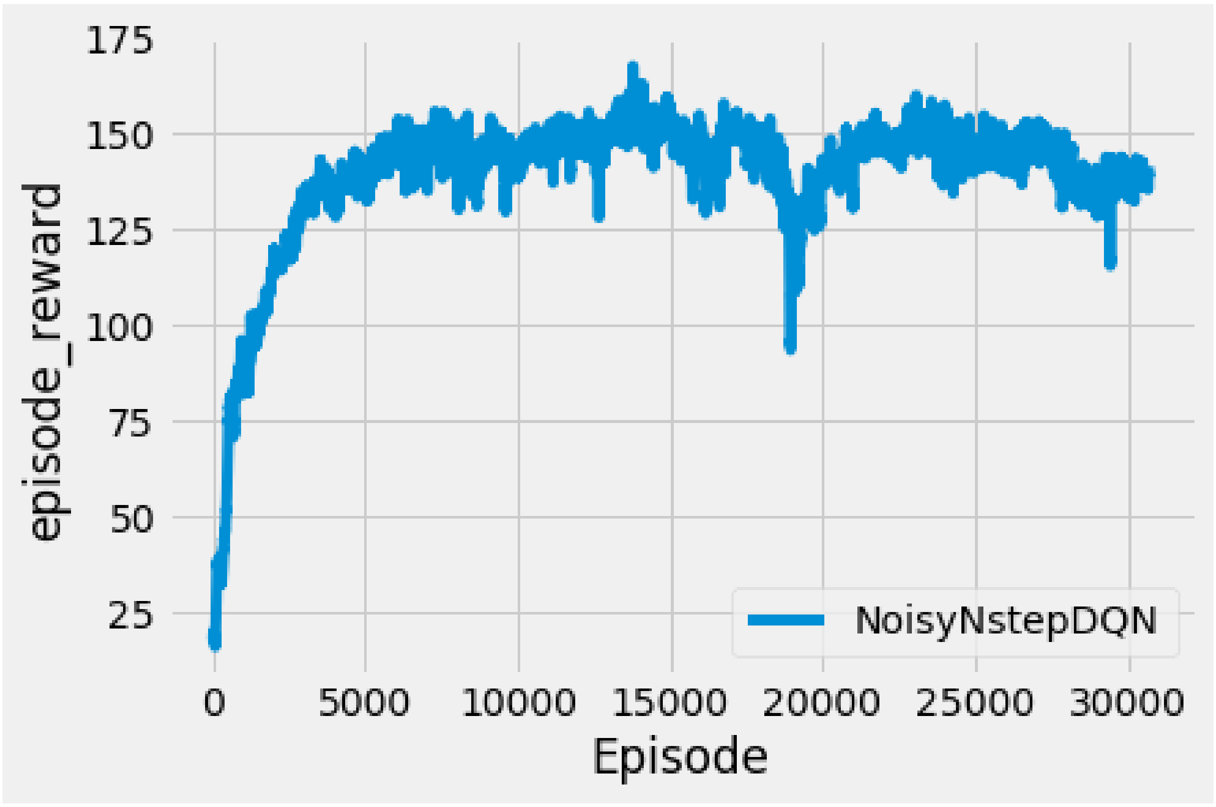


FIGURE 16 – The reward function by episode

This last extension go a long way towards improving both overall performance and sample efficiency with a wire-to-wire victory over both previous versions. We left NoisyNstepDQN running for a total of 30 million steps.
We're now consistently clearing the first level, and pulling off tight maneuvers between ghosts while navigating towards left-over dashes with a purpose and making good use of the shortcut exits on each side of the screen.

Conclusion

Reinforcement Learning has emerged as a hot topic with both industrial and academic interest in a way that makes AI experts nowadays believe that the this approach is the most viable strategy for achieving human or superhuman Artificial General Intelligence (AGI).

As for our case, our work consisted basically of exploring and applying knowledge related to Reinforcement Learning. As a matter of a fact, through our research study, we tried first to explain the Markov Decision Process as the classic tool used for decision-making in reinforcement learning problems. In a second place, we detailed the different approaches used in RL, mainly the model-based methods, the value learning and the policy-based methods. All along these chapters, we listed the major algorithms of each approach and we explained the use case of every algorithm.

In addition to the research study, we put into practice all these theoretical knowledge's by implementing an agent that can play MsPacman using Python and the library Gym.

As every project can be enhanced and expanded, various perspectives are imminent for our work. As for research study, we can detail the concept of Hierarchical reinforcement learning (HRL) as a computational approach that intends to address the limitation of reinforcement learning to operate on different levels of temporal abstraction.
As for our application, we can train the same agent to learn to play other Atari games and successfully learn control policies directly from high-dimensional sensory input using reinforcement learning.

It should be underlined that our work represents only the tip of the iceberg of reinforcement learning. In fact, even though video games and Atari are a well-known benchmark for RL, they are not the only application areas of it. RL could be used and is applied in various domains such as finance, robotics, health care, natural language processing... By the end of our project, we would have liked to apply the same algorithm we used with MsPacman on a natural language translation application, but owing to lack of time we weren't able to adapt the algorithm and train it.

Références

[1] Algorithms for Reinforcement Learning by Csaba Szepesvari :
https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs-lecture.pdf

[2] RL Course by David Silver :
https://www.youtube.com/watch?v=2pWv7GOvuf0&list=
\PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9-

[3] Components of a Reinforcement Learning Agent :
https://becominghuman.ai/components-of-an-rl-agent-and-its-application\
-on-snake-1b3b6c8e1de5

[4] Reinforcement Learning : An Introduction by Richard S. Sutton and Andrew G. Barto
http://incompleteideas.net/book/bookdraft2017nov5.pdf

[5] Playing Atari with Deep Reinforcement Learning, Volodymyr Mnih and A.l DeepMind Technologies
https://arxiv.org/abs/1312.5602

[6] Dueling Network Architectures for Deep Reinforcement Learning, Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas
https://arxiv.org/abs/1511.06581

[7] Noisy Networks for Exploration, Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg
https://arxiv.org/abs/1706.10295

[8] keras-rl, Matthias Plappert, 2016, GitHub
https://github.com/keras-rl/keras-rl