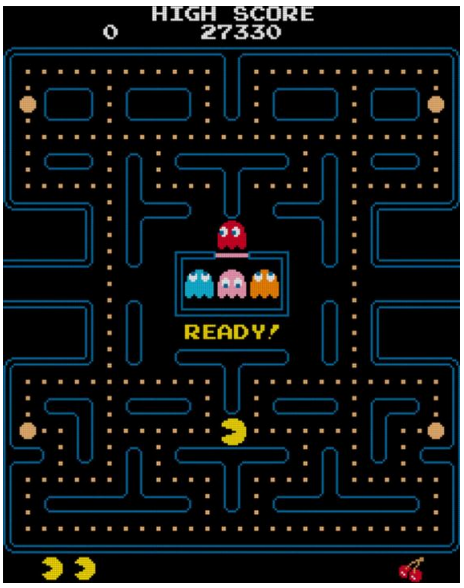
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Projet de Fin d'études

# Apprentissage par renforcement via l'environnement GYM




Lucas POTIN

# Contents

# Chapter 1

# Introduction

L'apprentissage par renforcement est une technique d'apprentissage qui s'est popularisée le siècle dernier notamment dans le domaine des Jeux. Nous pouvons par exemple citer AlphaZero, version généraliste d'AlphaGo adaptée au Go,Échecs et Shogi, qui utilise des concepts d'apprentissage par renforcement et présente des résultats convaincants.

J'ai pour ma part souhaité reprendre le sujet de Louis BAGOT, partant de l'article de 2013 de DeepMind [1]. Toutefois, je me suis d'abord concentré sur des méthodes et des jeux plus simple en premier lieu, pour apprivoiser les concepts de l'apprentissage par renforcement, avant de passer ensuite sur une application aux jeux Atari.

Dans ce rapport, nous trouverons dans un premier temps une partie théorique, présentant les bases de l'apprentissage par renforcement ainsi que les concepts clefs de ce domaine. Les parties suivantes seront à chaque fois dédiées à un ou plusieurs jeux, en indiquant les modifications/ajouts nécessaires pour appliquer l'apprentissage par renforcement, et en présentant quelques un de mes résultats. Enfin, une ouverture sera proposée sur l'utilité de l'apprentissage par renforcement, et sa force pour résoudre certains problèmes actuels.

# Chapter 2

# Concepts de base de l'apprentissage par renforcement

## 2.1 Principe

L'apprentissage par renforcement consiste à faire évoluer un agent dans un environnement. Une description des caractéristiques de l'environnement à un instant t est nommé État. L'agent peut interagir sur l'environnement via des actions et reçoit, en réalisant ces dernières une récompense, qui peut être positive ou négative. Ce cycle peut être résumé par le schéma suivant :

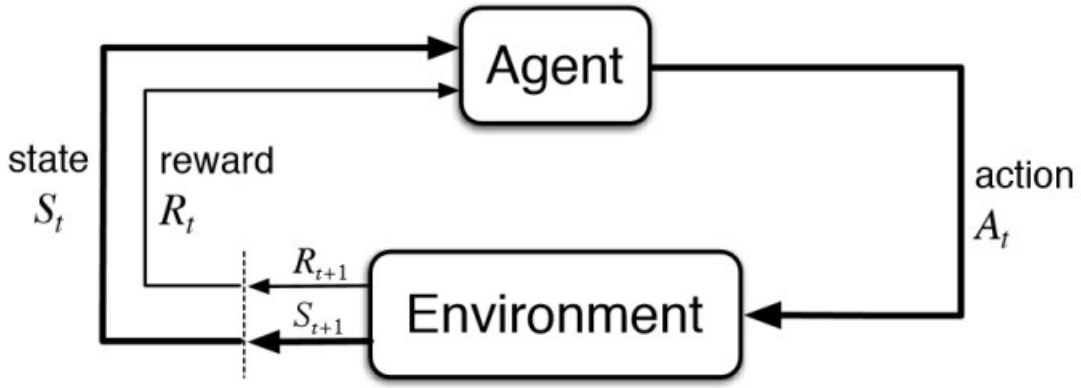



Figure 2.1: Cycle de l'apprentissage par renforcement

Le but de l'apprentissage par renforcement est de déterminer une politique, c'est à dire une fonction $\pi$ retournant une probabilité sur l'ensemble d'action selon un état. Nous souhaiterons alors une politique optimale, c'est à dire qui va maximiser les récompenses potentielles.

Dans la suite du rapport, nous utiliserons les notations suivantes:

- S l'ensemble des états
- A l'ensemble des actions
- R l'ensemble des récompenses
- P l'ensemble des probabilités de transitions : on notera $P(s, a, s') \in (0, 1)$ la probabilité d'arriver dans l'état $s'$, ayant réalisé l'action a dans l'état s.

## 2.2 Prédiction et Contrôle

L'apprentissage par renforcement se décompose en deux étapes, notées prédiction et contrôle.

- L'agent doit être capable, connaissant une politique d'évaluer son efficacité : c'est ce que nous appellerons la **prédiction**
- Il doit être également capable de trouver la politique optimale , c'est à dire celle qui va maximiser les récompenses potentielles : c'est ce que nous appellerons le **contrôle**

## 2.3 Récompense et retour

Dans le cadre de l'apprentissage par renforcement, à un instant t, l'agent reçoit une récompense immédiate que nous noterons $r_t$. Toutefois, lorsqu'on cherche une stratégie optimale dans un jeu, on souhaite considérer à la fois les récompenses à court terme, mais également à long terme. En effet, un coup n'apportant rien à un instant t peut être décisif à un instant t+n. L'objectif de l'agent va alors être de maximiser le cumul des récompenses possibles. Dans ce contexte, nous allons noter $R_t$ le retour potentiel , avec :

$$R_t = r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1}$$

Cette notion de retour potentiel permet d'estimer la somme des récompenses potentielles, pondérées par un facteur $\gamma$, nommé facteur d'actualisation. Ce dernier est compris entre 0 et 1 :

- S'il vaut 0 , on adopte une approche à court terme, et on ne considère que les récompenses immédiates
- S'il vaut 1, on donne le même poids à toutes les récompenses.

## 2.4 Méthodes values based

En apprentissage par renforcement, il existe deux grands types de méthodes : les méthodes **value-based** et les méthodes **policy-based**. Nous nous intéresserons aujourd'hui uniquement aux premières. Dans les méthodes value-based, nous allons introduire le concept d'une fonction de valeur, notée $V(s)$, et représentant l'intérêt d'être dans un certain état. Nous posons alors :

$$V^{\pi}(s) = \mathbb{E}[R_t | s_t = s]$$

V représente alors le retour espéré lorsque nous nous situons dans un état s, suivant une politique $\pi$. Cette formulation, bien qu'intuitive, ne montre par explicitement l'impact des actions. Nous introduisons alors une deuxième fonction de valeur, notée $Q(s, a)$, avec :

$$Q^{\pi}(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a]$$

L'idée est alors la même , à la différence qu'on se place à la fois dans un état s, mais également comme ayant réalisé une action a.

L'idée des méthodes value-based va alors d'être capable d'estimer ces fonctions de valeurs. En effet, si les fonctions de valeurs sont connues, la politique optimale devient triviale à déterminer, puisqu'il suffit de prendre les valeurs les plus importantes, qui vont donc maximiser le retour espéré.

Nous pouvons alors résumer le fonctionnement d'une méthode value based via le schéma suivant :

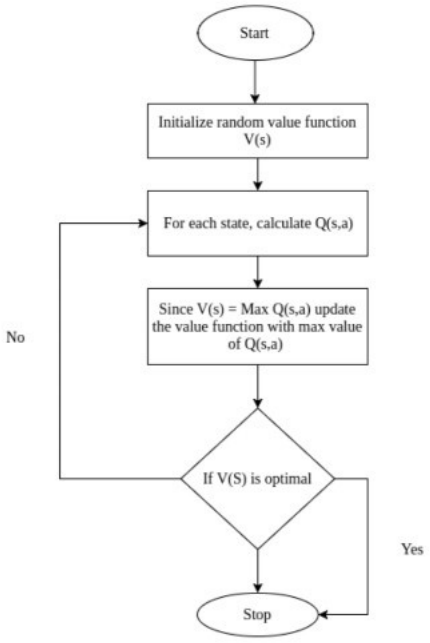



Figure 2.2: Apprentissage value-based

Une méthode value-based s'occupe dans la plupart des cas conjointement des phases de prédiction de contrôle : en prédisant la fonction de valeur, l'algorithme peut mettre à jour la politique recherchée via un opérateur max et la rendre meilleure, puis à son tour prédire la fonction sur cette nouvelle politique. La politique optimale est donc approchée étape par étape.

## 2.5 Équations de Bellman

Nous venons de définir $V^{\pi}(s)$ via l'espérance sur le retour potentiel depuis l'état s. Nous pouvons alors introduire l'équation de Bellman, permettant de déterminer V(s) :

$$V(s) = max_a P(s, a, s^{'})(R(s, a) + \gamma V(s^{'}))$$

Dans un environnement déterministe, c'est à dire où nos probabilités de transitions sont booléennes, $V(s)$ peut être calculé via l'équation de Bellman :

$$V(s) = max_a(R(s, a) + \gamma V(s^{'}))$$

De manière analogue, nous avons pour Q :

$$Q(s, a) = max_a(R(s, a) + \gamma Q(s^{'}, a))$$

## 2.6 Exploration/Exploitation

Comme vu précédemment, l'agent va devoir choisir des actions pour interagir avec l'environnement. S'il est évident de choisir la bonne action lorsque nous connaissons la politique optimale, un choix doit être fait pendant l'apprentissage. En effet, nous avons deux possibilités :

- Choisir une action aléatoire
- Choisir une action en fonction des connaissances déjà acquises

Lorsque nous choisissons une action aléatoire, nous sommes dans l' **exploration**. Ceci va nous permettre de découvrir de nouvelles expériences, et ainsi d'agrandir nos connaissances. Toutefois, trop d'exploration ne peut conduire qu'à une non convergence, puisque jouer purement aléatoirement n'est pas une manière convenable de jouer dans la plupart des jeux.

Lorsque nous choisissons une action en fonction de nos connaissances, nous sommes dans l' **exploitation**. Ici, les avantages et inconvénients sont inversés : l'apprentissage va pouvoir converger, mais trop d'exploitation risque de le faire converger vers un minimum local, sans avoir eu l'occasion de visiter les expériences les plus profitables.

L'apprentissage par renforcement admet donc le dilemme de l'exploration/exploitation : il va être nécessaire de s'assurer de visiter un grand nombre d'états/actions, notamment au début du jeu, lorsque nous avons peu d'informations, pour ensuite utiliser de manière plus prononcée nos connaissances à la fin de l'apprentissage, lorsque nous sommes mieux informés.

## 2.7 On/Off Policy

Le but de l'apprentissage va être de déterminer une politique optimale, c'est à dire une fonction qui, à un état associé va indiquer l'action à réaliser. Cette politique peut être purement déterministe, ou stochastique : en effet, dans le cas du jeu pierre-papier-ciseau par exemple, une politique optimale ne peut qu'être stochastique. Dans l'apprentissage par renforcement, nous avons besoin en réalité d'utiliser une politique lors de deux étapes :

- Lors du choix de l'action : c'est ce que nous appellerons la **Behaviour policy**
- Lors de la mise à jour des valeurs de $V(s)$ ou $Q(s, a)$ : c'est ce que nous appellerons la **Update policy**

Il existe alors deux types d'algorithmes :

- Si la même politique est utilisée pour ces deux étapes, l'algorithme est dit **On policy**
- Si ce sont deux politiques différentes, l'algorithme est dit **Off policy**

Ce choix va avoir un impact sur le comportement lors de l'apprentissage, nous développerons ce point dans le chapitre suivant.

# Chapter 3

# Premier pas : les méthodes tabulaires

## 3.1 Jeux associés

### 3.1.1 GridWorld

Dans cette partie, nous allons proposer un jeu simple, que nous nommerons GridWorld.

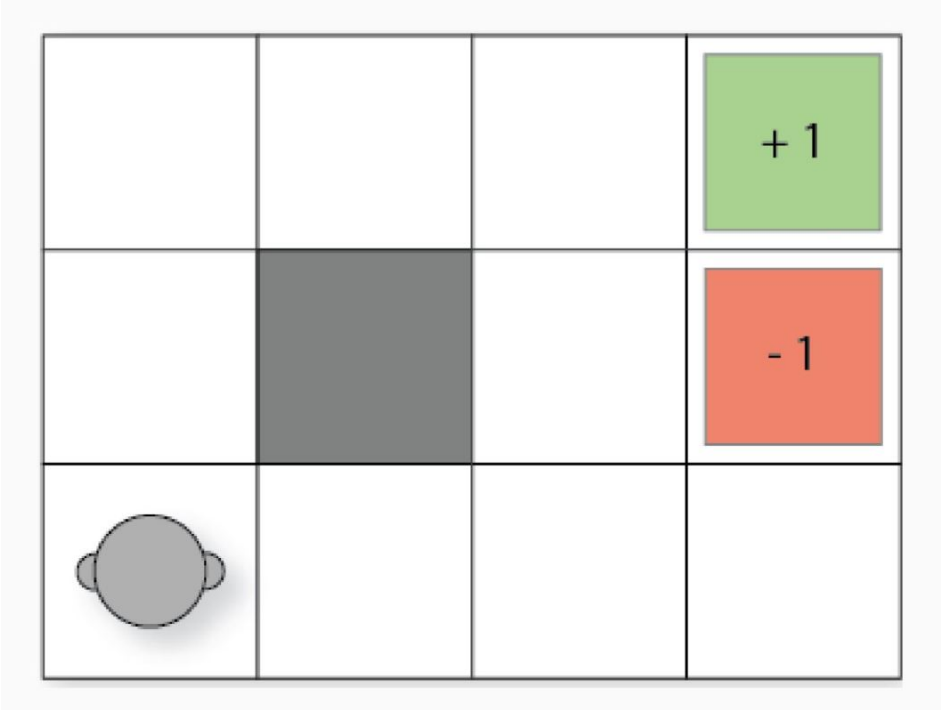



Figure 3.1: GridWorld

Un agent se déplace dans un labyrinthe, représenté sous la forme d'une grille. Il peut se déplacer dans les quatre directions. Le jeu est considéré comme terminé quand l'agent arrive dans une case de couleur : il a gagné si il arrive sur la case verte, et perdu si il arrive sur la case rouge. Le labyrinthe est composé de cases blanches, où l'agent peut aller, et de cases noires, représentant des murs. Si l'agent tente d'effectuer une action qui le ferait sortir du labyrinthe ou aller dans un mur, l'action n'est pas considérée.

## 3.2 Les méthodes tabulaires

En analysant ce jeu du GridWorld, nous pouvons observer qu'il est composé au maximum de NxM états , avec N le nombre de lignes de la grille et M le nombre de colonnes et de 4 actions. Par conséquent, en reprenant l'idée d'une fonction de valeurs états/actions, nous n'avons en réalité que NxMx4 valeurs possibles. Nous pouvons alors stocker ces valeurs dans une matrice : c'est l'idée des méthodes tabulaires.

### 3.2.1 Q-Learning

Il est alors temps d'introduire un premier algorithme, le Q-Learning qui va se décomposer sous la forme suivante :

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize $s$
  Repeat (for each step of episode):
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $a$, observe $r$, $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until $s$ is terminal

Figure 3.2: Algorithme du Q-learning

Pour mieux comprendre cet algorithme , regardons le schéma suivant :

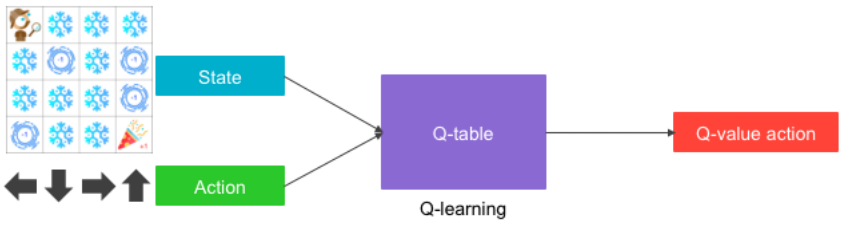



Figure 3.3: Q-Learning et Matrice

Étant donné un état s et une action a , nous allons pouvoir récupérer dans notre matrice la valeur $Q(s, a)$ associée. Nous pouvons également appliquer ce raisonnement dans le sens inverse : étant donné un état s, et ayant la possibilité de récupérer $Q(s, a)$ pour toutes les actions, nous allons pouvoir récupérer l'action qui maximise $Q(s, a)$.

Au début, nous ne connaissons évidemment pas les valeurs de notre matrice Q. Elle est donc initialisée de manière arbitraire. L'apprentissage va alors consister en la répétition d'épisodes ( représentant une partie ), eux mêmes décomposés en étapes (action par action). Pour chaque étape, le raisonnement va alors être le suivant:

- Dans un premier temps, une action est choisie, via une politique $\epsilon$-Gloutonne.
- Cette action est réalisée sur l'environnement, qui renvoie la récompense associée , ainsi que le nouvel état.
- La récompense est utilisée pour mettre à jour les Q-Values.

Précisons maintenant le terme de politique $\epsilon$-Gloutonne. Nous allons fixer un paramètre $\epsilon$ qui va décroître avec le temps. Ce paramètre commencera à 1, et diminuera jusqu'à une valeur $\epsilon_{max} > 0$. A chaque étape, nous ferons un tirage de variable aléatoire $p$ sur une loi uniforme dans $(0, 1)$, et nous appliquerons le schéma suivant :

- Si $p < \epsilon$, nous choisissons une action aléatoirement : nous sommes donc dans le cadre de l'exploration
- Si $p > \epsilon$, nous choisissons une action en prenant l'argument maximal de la Q-Value : nous sommes donc dans le cadre de l'exploitation

Les Q-Values sont alors calculées via la formule suivante :

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

Figure 3.4: Formule de mise à jour des Q-values

Cette formule réutilise les équations de Bellman introduites précédemment, en rajoutant un hyperparamètre, $\alpha$ compris entre 0 et 1, nommé facteur d'apprentissage. Ce dernier va représenter la vitesse d'apprentissage. Plus il est grand, plus les modifications des Q-Values vont être fortes, mais plus le risque de divergence est grand. Dans le cadre du Q-Learning, les Q-values sont mises à jour en utilisant un opérateur max. Cela est donc comparable à une politique complètement Gloutonne. Les deux politiques de décision et de mise à jour sont différentes, c'est pourquoi le Q-Learning est un algorithme "off-policy".

### 3.2.2 SARSA

Présentons maintenant un second algorithme, nommé SARSA, pour **S**tate, **A**ction, **R**eward, **S**tate, **A**ction.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize $s$
  Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Repeat (for each step of episode):
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-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 3.5: Algorithme SARSA

Le principe de l'algorithme reste celui du Q-Learning, la seule différence se fait au niveau de la mise à jour des Q-Values. Dans SARSA, une fois l'action réalisée, nous tirons une nouvelle fois une action, et c'est cette dernière qui va être utilisée dans la formule de mise à jour. Il n'y a donc ici pas d'opérateur max. Les deux politiques de décision et de mise à jour sont identiques, c'est pourquoi le Q-Learning est un algorithme "on-policy".

## 3.3 Résultats

### 3.3.1 Q-Learning

Nous allons utiliser la version du GridWorld présentée en introduction de ce chapitre :

- L'agent commence sur la case 0 0
- La case de victoire est à la position 2 3
- la case de défaite est à la position 1 3
- il y a un mur en position 1 1

Nous obtenons alors à la fin de notre entraînement la Q-Table suivant, pour 100 episodes :

| État (PosJoueur) | Haut | Gauche | Bas | Droite |
|---|---|---|---|---|
| (0,0) | 0.956 | 0.0521 | 0.0522 | 0.0696 |
| (1,0) | 0.966 | 0.105 | 0.0501 | 0.129 |
| (2,0) | 0.066 | 0.073 | 0.018 | 0.977 |
| (0,1) | 0.039 | 0.032 | 0.039 | 0.140 |
| (1,1) | 0 | 0 | 0 | 0 |
| (2,1) | 0.102 | 0.017 | 0.063 | 0.998 |
| (0,2) | 0.310 | 0.019 | 0.048 | 0.004 |
| (1,2) | 0.633 | 0.032 | 0.017 | -0.307 |
| (2,2) | 0.062 | 0.022 | 0.013 | 0.999 |
| (0,3) | -0.431 | 0.027 | 0.003 | 0.003 |
| (1,3) | 0 | 0 | 0 | 0 |
| (2,3) | 0 | 0 | 0 | 0 |

Une fois notre table obtenue, nous pouvons dégager la politique optimale associée en prenant simplement pour chaque état ( c'est à dire chaque position de l'agent) l'action avec la Q-Value la plus élevée. Nous obtenons alors la politique optimale suivante :

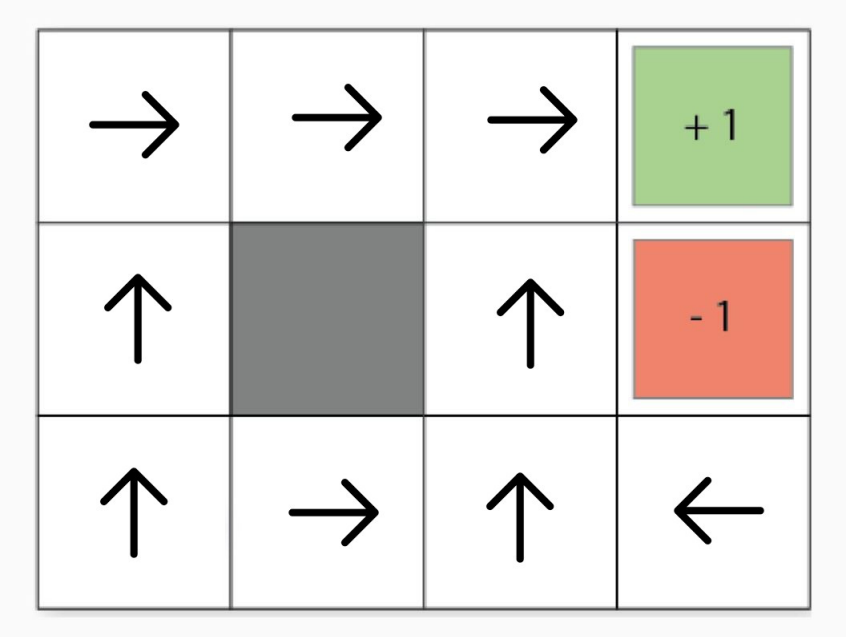



Figure 3.6: Politique Q-Learning pour le GridWorld

### 3.3.2 SARSA

Notre implémentation de SARSA trouve la même politique optimale, pour un Q-table suivante :

| Etat (PosJoueur) | Haut | Gauche | Bas | Droite |
|---|---|---|---|---|
| (0,0) | 0.935 | 0.0521 | 0.0522 | 0.0696 |
| (1,0) | 0.976 | 0.205 | 0.001 | 0.029 |
| (2,0) | 0.036 | 0.074 | 0.008 | 0.947 |
| (0,1) | 0.059 | 0.012 | 0.019 | 0.152 |
| (1,1) | 0 | 0 | 0 | 0 |
| (2,1) | 0.102 | 0.017 | 0.063 | 0.998 |
| (0,2) | 0.310 | 0.019 | 0.048 | 0.004 |
| (1,2) | 0.633 | 0.032 | 0.037 | -0.417 |
| (2,2) | 0.074 | 0.022 | 0.013 | 0.999 |
| (0,3) | -0.531 | 0.021 | 0.002 | 0.004 |
| (1,3) | 0 | 0 | 0 | 0 |
| (2,3) | 0 | 0 | 0 | 0 |

### 3.3.3 Choix de la méthode

Sur notre jeu simple, les performances sont sensiblement identiques. Toutefois, des recherches ont montré que le Q-Learning permet de trouver la solution la plus optimale, au contraire de SARSA, qui est plus prudent au niveau de l'apprentissage, et parfois seulement converger vers un maximum local. Toutefois, cela implique que dans le cadre du Q-Learning, pendant l'apprentissage, l'agent va déclencher les récompenses négatives coûteuses. Dans la suite de ce PFE, nous travaillerons sur des jeux, le fait de perdre ne sera donc pas très grave. Nous pouvons alors utiliser le Q-Learning sans risque. Toutefois, dans d'autres domaines, par exemple de la robotique, il serait possible qu'une mauvaise action déclenche par exemple la destruction du robot. Il faudrait alors envisager des politiques plus sûres au niveau de l'apprentissage, et SARSA deviendrait peut être une meilleure solution

# Chapter 4

# Apprentissage via Réseau de Neurones

## 4.1 Jeux associés

A partir de maintenant, nous allons pouvoir utiliser la librairie Gym, fournie par OpenAI. Cette librairie fournit un environnement pour de nombreux jeux, notamment des versions de Blackjack, mais également de jeux plus compliqués comme des Jeux Atari.

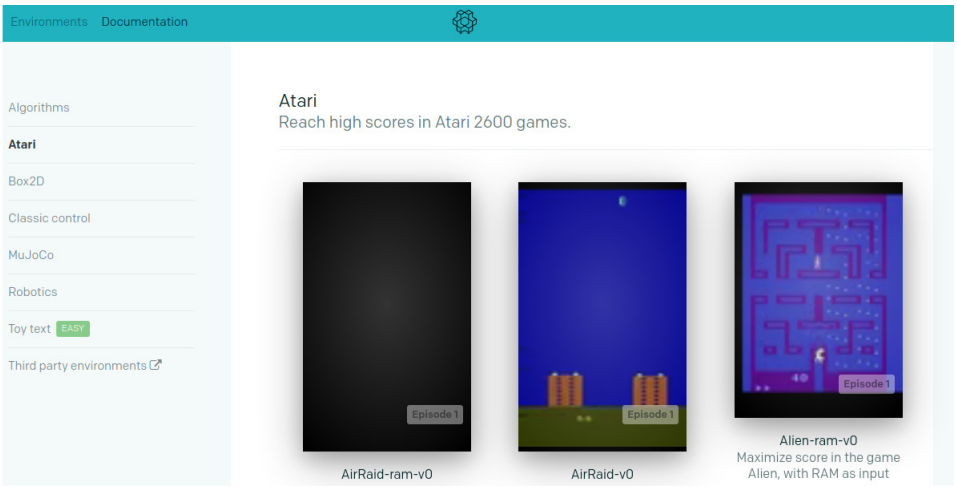



Figure 4.1: Interface du site GYM

Cette librairie va permettre d'automatiser la partie "Jeu" dans nos expériences : nous n'aurons plus qu'à fournir l'action à réaliser à chaque étape, et GYM nous renverra, via une méthode step, la récompense associée, ainsi que le nouvel état du Jeu. Par conséquent, nous n'aurons plus à définir les récompenses, états et actions de nos jeux, puisque Gym s'en occupera pour nous. Gym donnant accès à un nombre de jeux conséquent, nous pourrons ainsi étudier plusieurs jeux de manière plus aisée.

### 4.1.1 LunarLanding

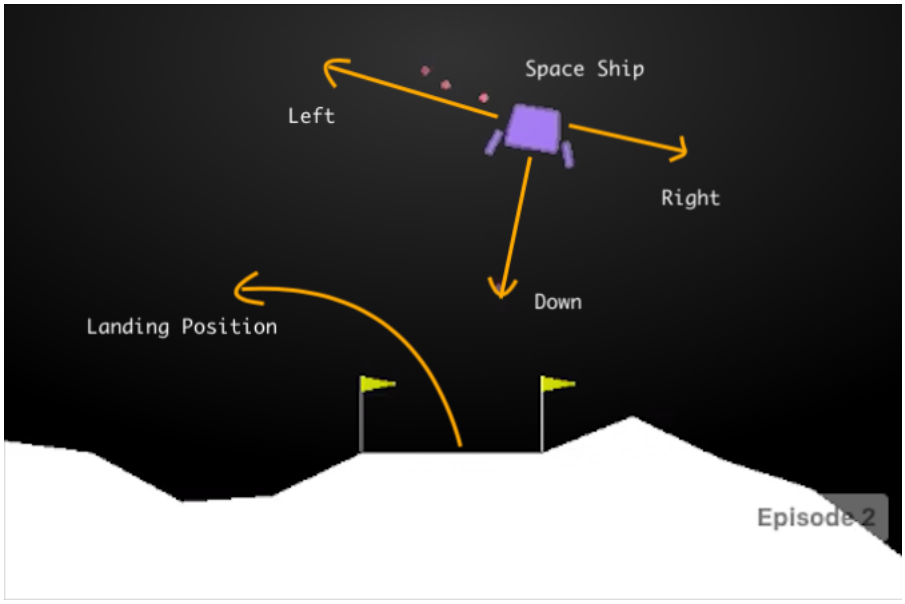



Figure 4.2: Lunar Landing

Dans LunarLanding, le joueur doit faire atterrir une navette spatiale entre deux drapeaux jaunes. Il peut effectuer 4 actions :

- Utiliser le réacteur droit
- Utiliser le réacteur central
- Utiliser le réacteur gauche
- Ne rien faire

Le jeu est alors composé de 3 types de récompenses :

- Entre 100 et 140 points pour le fait de descendre au sol
- 10 points par jambe de la navette touchant le sol
- un bonus de 100 points si la navette se pose convenablement et un malus de 100 points s'il y a crash

Dans LunarLanding, un état est représenté par un vecteur de 8 composantes, avec la position, la vitesse, des angles et l'information du contact des jambes avec le sol.

## 4.2 Apprentissage avec Réseau de Neurones

### 4.2.1 Motivation

Avec ce nouvel environnement, nous remarquons que notre nombre d'états a été démultiplié. En effet, un état correspond maintenant à un vecteur unique. Les composantes du vecteurs étant de nature continue, puisque réelles, notre ensemble d'état est également continu. Il devient donc impossible de fonctionner avec des méthodes tabulaires. C'est alors ici que le réseau de neurones apparaît. En effet, l'utilisation d'un réseau va nous permettre d'approximer nos Q-values, et va donc remplacer notre matrice.

### 4.2.2 Le réseau

LunarLanding est un jeu restant assez simple, nous allons alors proposer un réseau composé de couches Denses (Fully Connected). Notre réseau sera alors de la forme suivante :

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_31 (Dense)             (None, 150)               1350
_________________________________________________________________
dense_32 (Dense)             (None, 120)               18120
_________________________________________________________________
dense_33 (Dense)             (None, 4)                 484
=================================================================
Total params: 19,954
Trainable params: 19,954
Non-trainable params: 0
```

Figure 4.3: Modèle pour le jeu Lunar Landing

### 4.2.3 Fonctionnement du réseau

En entrée du réseau, nous donnerons le vecteur d'états, de 8 composantes. En sortie, nous récupérerons 4 valeurs, représentant l'approximation des Q-Values dans l'état actuel pour chaque action. Nous pourrons alors récupérer l'argument maximal de ces valeurs, pour trouver l'action optimale selon le réseau. Il reste maintenant à déterminer comment mettre à jour les poids de notre réseau, ce qui revient à déterminer comment calculer notre loss. Nous allons alors définir la loss de la manière suivante :

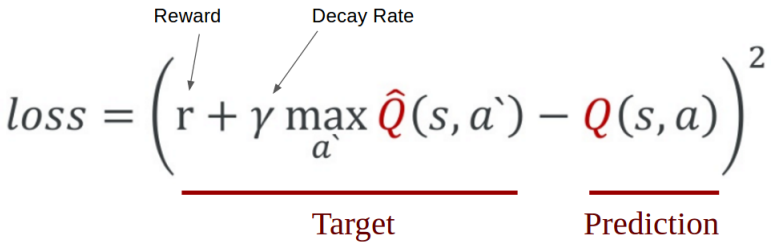



Figure 4.4: Loss en apprentissage par renforcement

Nous pouvons ici voir deux termes :

- Prédiction : c'est la valeur prédite par le réseau
- Target : C'est la valeur obtenue via les formules classiques du Q-learning, c'est à dire notre nouvelle Q-Value calculée

Notre loss va donc consister grossièrement en une forme quadratique de la différence entre l'ancienne et la nouvelle valeur de la Q-value, pour un état et une action choisie. Il suffira alors de faire une backpropation du gradient, pour mettre à jour nos poids.

### 4.2.4 Nouvel algorithme

Notre algorithme se présentera alors comme ceci :

- Initialisation du réseau et des paramètres
- Puis, pour chaque épisode, l'épisode est décomposé en étapes de la forme suivante :
  - Choix d'une action, de manière aléatoire ou avec l'aide du réseau
  - GYM renvoie la récompense et le nouvel état associé
  - Calcul de la loss
  - Mise à jour des poids

## 4.3 Résultats

Voici les résultats obtenus pour un apprentissage sur environ 400 épisodes, pour une durée d'environ 10 minutes :

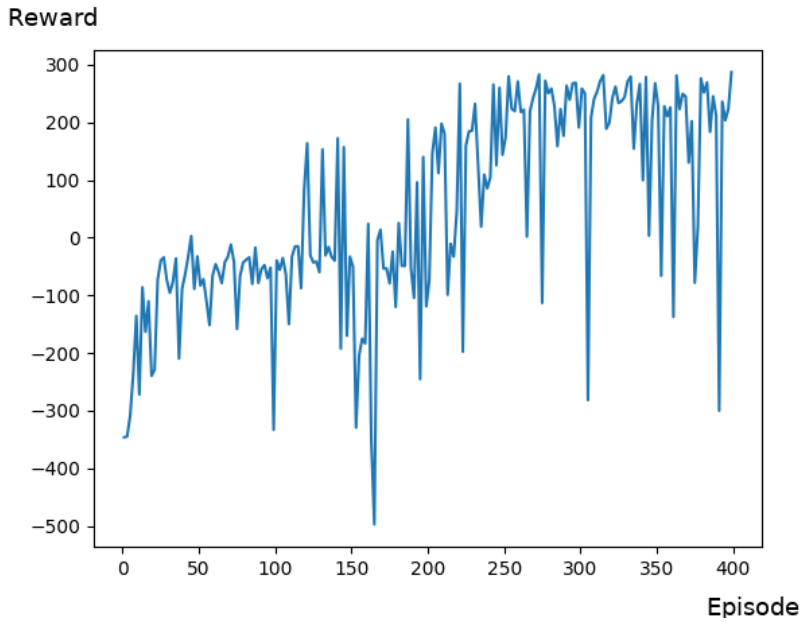



Figure 4.5: Récompenses pour LunarLanding

Nous pouvons alors voir que les récompenses semblent augmenter avec les épisodes, mais sont très oscillantes. Ceci est dû au fait que le jeu possède une composante très punitive, avec le bonus/malus selon s'il y a crash ou non. Puisque notre politique de choix d'action n'est jamais purement gloutonne, car un $\epsilon$ minimal est fixé lors de l'apprentissage, il est possible que l'agent fasse une action aléatoire qui provoque un crash, même au bout de plusieurs épisodes. Nous décidons alors d'afficher plutôt la valeur des moyennes des récompenses en fonction des épisodes, pour mieux saisir l'évolution

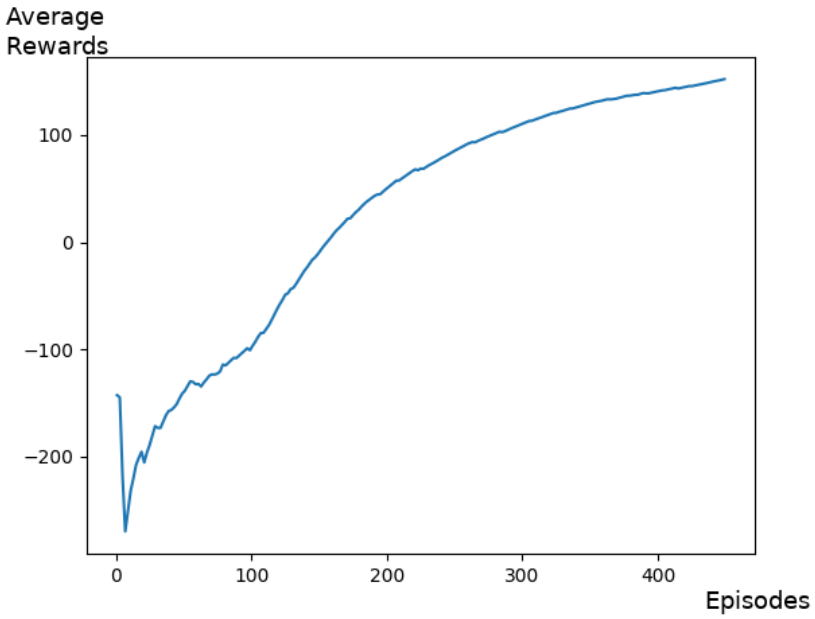



Figure 4.6: Recompenses moyennes pour LunarLanding

Ici, nous pouvons voir de manière nette l'apprentissage, avec la moyenne des récompenses qui augmente bien avec les épisodes. A la fin de l'apprentissage, l'agent sait effectivement jouer, et se pose en douceur entre les drapeaux

# Chapter 5

# Jeux Atari : Le Deep Q Learning

## 5.1 Jeux associés

### 5.1.1 Dilemme Ram vs Images

Dans toute la suite de ce Projet, nous allons travailler sur des jeux Atari. Pour ces derniers, GYM propose deux versions des jeux, influant sur la manière dont est représenté un état :

- Dans les versions RAM, un état est représenté par un bloc de 128 bytes
- Dans les versions Image, un état est représenté par une image, en général de dimension 210x160x3.

Ce projet utilisera les versions Images, et ceci pour deux raisons :

- La plupart des travaux consultables sur Internet étaient basé sur cette version
- Ayant vu les couches de convolution en Deep Learning, j'ai souhaité travailler sur des images pour directement mettre ces connaissances à profit.

### 5.1.2 Pong

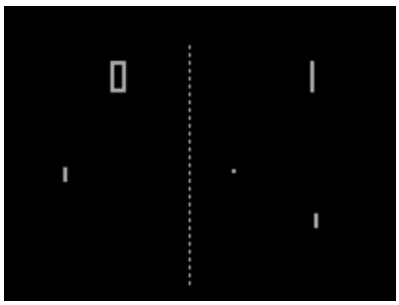


Figure 5.1: Pong

Dans Pong, le joueur doit renvoyer une balle vers l'adversaire via une raquette représentée par une barre. Il dispose de deux actions possibles, aller en haut ou aller en bas. Le joueur gagne 1 point quand l'adversaire ne parvient pas à renvoyer la balle, et en perd un quand il ne parvient pas à renvoyer la balle. Un épisode dure jusqu'à ce qu'un joueur arrive à 21 points.

### 5.1.3 Pacman

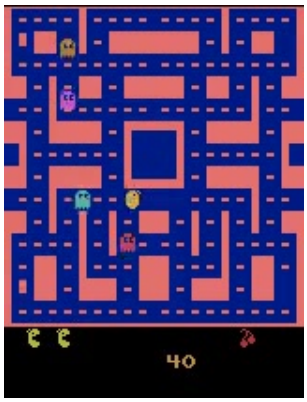



Figure 5.2: Pacman

Dans Pacman, Pacman se déplace en essayant de manger le maximum de Pac-gommes tout en esquivant les fantômes. Il peut se déplacer dans les quatre directions. Manger des pac-gommes rapporte des points, tandis que percuter un fantôme fait perdre une vie. L'épisode s'arrête quand Pacman a mangé toutes les gommes ou quand il a perdu ses 3 vies

### 5.1.4 Breakout

Figure 5.3: Breakout

Dans Breakout, le joueur doit casser le maximum de briques tout en gardant la balle dans l'écran. Casser des briques rapporte un certain nombre de points, fixes selon la couleur de la brique. L'épisode se termine quand toutes les briques sont cassées ou quand le joueur a perdu toutes ses vies

### 5.1.5 Enduro

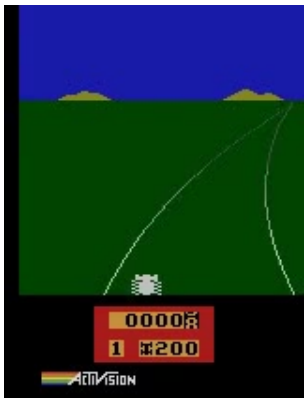



Figure 5.4: Enduro

Dans Enduro, le joueur conduit une voiture, et doit réaliser un tour de circuit en esquivant les voitures adverses. Percuter une voiture fait reculer le joueur.

## 5.2 Le Deep Q Learning

Ces nouveaux jeux impliquent d'introduire de nouveaux concepts, car nous ne pouvons pas simplement utiliser la méthode précédente en changeant juste l'environnement GYM. Dans un papier de 2013 [1], Volodymyr Mnih propose un algorithme, nommé le Deep-Q-Learning, qu'il décide d'appliquer aux jeux Atari. Nous allons alors présenter les différents changements et améliorations proposés.

### 5.2.1 Preprocessing

En utilisant les jeux Atari comme environnement, nous disposons maintenant d'images comme représentation d'états, de taille 210x160x3. Il va alors être nécessaire d'effectuer un preprocessing de ces images, avant de les utiliser pour notre réseau de neurones.

- Dans un premier temps, nous pouvons transformer nos images en noir et blanc. En effet, la couleur n'apporte pas d'information utile dans les jeux Atari présentés. Nos images deviennent donc de taille 210x160x1
- Nous pouvons ensuite rogner nos images, pour réduire encore la taille. Par exemple , dans Pong, nous n'avons pas besoin d'avoir la partie derrière les barres des joueurs. Nous pouvons donc la supprimer, ce qui va alléger la taille des entrées pour le réseau , et ainsi le nombre de paramètres.

- Dans un dernier lieu, nous allons effectuer un rescale, pour transformer nos images finales au format 84x84x1

En plus de ce traitement des images, il va être nécessaire d'effectuer un stacking. En effet, une seule image ne donne pas d'indication sur le déplacement, or ceci est une composante indispensable dans nos jeux. Si nous prenons deux images, nous avons une notion de vitesse, mais pas d'accélération, notion également importante. Nous allons alors fixer une variable $Stack_Frame$, correspondant au nombre de frame que nous allons considérer comme représentant un état. Dans plusieurs papiers, il est conseillé d'utiliser 4 frames à chaque fois, nous reprendrons donc cette valeur. A la fin de ce preprocessing, nous obtenons alors des groupes de 4 frames, chacune de taille 84x84. Ce sont ces groupes qui vont représenter un état pour notre algorithme, et non pas les images de taille 210x160x3 initiales.

### 5.2.2 Experience Replay

L'utilisation d'un réseau de neurones comme approximateur provoque toutefois deux problèmes :

- Ce dernier à tendance à oublier certaines valeurs
- La corrélation temporelle des différentes paires états/actions dégradent l'apprentissage

Deepmind propose alors un système d'experience replay. Nous définirons alors une expérience comme un tuple **SASR**, c'est à dire :

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

Nous allons alors stocker ces expériences dans une mémoire, et tirer des batchs d'expériences depuis la mémoire pour entraîner le réseau. Utiliser une mémoire pour donner de manière aléatoire des batchs d'expérience, plutôt que les expériences les unes après les autres possède deux avantages :

- La convergence de l'approximation de la fonction de valeur sera meilleure, car les données sont identiquement et indépendamment distribuées
- En utilisant les expériences plusieurs fois, puisque réinjectées dans le réseau, nous pourrons réduire le nombre d'expériences nécessaires. Ici, nos expériences ne sont pas coûteuses, mais dans d'autres problèmes, cela peut être très intéressant.

### 5.2.3 Nouveau Réseau

Travaillant ici avec des images, nous allons utiliser non plus seulement des couches denses , mais également des couches de convolutions 2D. Ces dernières vont permettre de faire un balayage sur l'image, pour observer les corrélations spatiales. Le réseau va alors être de la forme suivante :

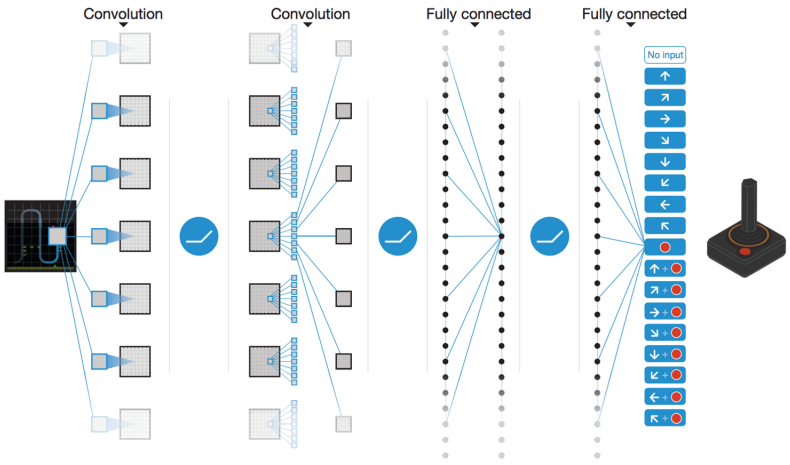



Figure 5.5: Description du réseau

En entrée, nous donnerons des batchs d'états. Le réseau sera alors composé de 4 couches de convolutions, suivies de 2 couches denses. En sortie, il y aura toujours $|A|$ valeurs , pour les Q-Values associées à chaque action.

### 5.2.4 Nouvel Algorithme

Avec ces nouvelles modifications, notre algorithme du Deep Q Learning deviendra alors de la forme suivante :

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

Figure 5.6: Algorithme Deep-Q-Learning

## 5.3 Résultats

Nous avons d'abord testé l'algorithme sur Pacman, pour les résultats suivants, au bout du 20 heures d'apprentissage

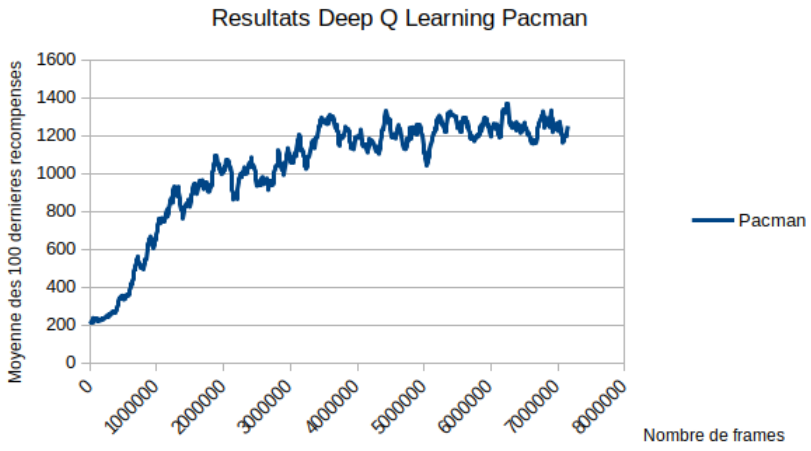



Figure 5.7: Resultats Deep Q Learning Pacman

En abscisse est indiqué le temps, ici représenté par le nombre de frames passées dans le réseau. En ordonnée, pour suivre la progression de l'apprentissage, nous afficherons la moyenne des 100 dernières récompenses. L'apprentissage est visible, toutefois, il semble stagner à partir d'un certain stade.

Voici également des résultats pour le jeu Breakout :

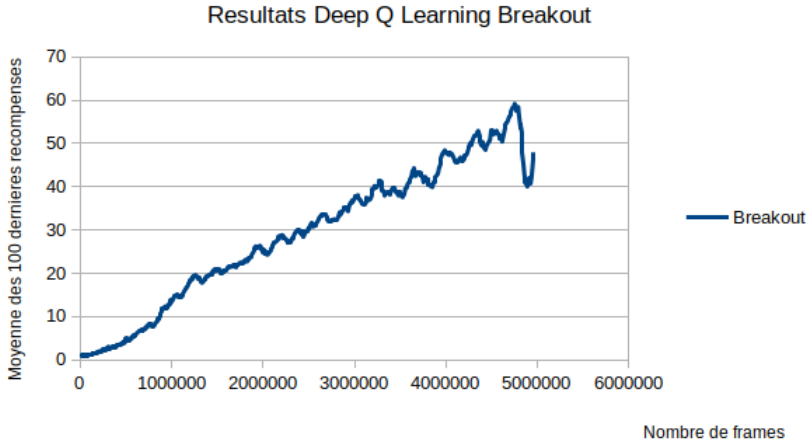



Figure 5.8: Resultats Deep Q Learning Breakout

# Chapter 6

# Ameliorations au Deep Q Learning

Le Deep Q Learning implémenté comme dans la partie précédente n'est pas très efficace, car il s'agit de la version de base , sans aucune amélioration. Il a été ensuite modifié durant les 8 dernières années, que ce soit pour améliorer sa vitesse d'apprentissage, mais également sa stabilité.

## 6.1 Prioritized Experience

Dans notre première implémentation du Deep Q Learning, nous avons introduit le concept d'Experience replay, qui consiste à stocker des tuples nommés expérience, qui sont stockés dans une mémoire, puis récupérés pour alimenter le réseau de neurones. Dans notre première implémentation, les expériences étaient sélectionnées de manière aléatoire pour être fournies en batch au réseau. Dans un papier de 2016[2], Tom Schaul propose alors d'introduire un nouveau attribut aux tuples, représentant la valeur de priorité. Cette dernière permet d'avoir un indicateur pour savoir si choisir cette expérience va apporter beaucoup ou peu au réseau. Il reste maintenant à savoir comment estimer cette priorité. Dans le chapitre 2 , nous avons indiqué que la loss, dans notre réseau, correspondait à la différence entre la valeur prédite par le réseau de la Q-value, et celle calculée via les équations de Bellman. Par conséquent , une loss élevée indique que l'action actuelle a beaucoup modifié la valeur de la Q-Value : ce sera donc une expérience intéressante, que nous voudrons réinjecter plusieurs fois dans le réseau. Notre valeur de priorité va donc directement être liée à la loss. Une fois cet indicateur calculé, nous pouvons maintenant poser $P(i)$, la probabilité que l'expérience i soit choisi dans le batch fourni au réseau de la forme suivante :

$$P(i) = \frac{p_i^\omega}{\sum_k p_k^\omega}$$

Ici , $\omega$ est un hyperparamètre entre 0 et 1, représentant la part d'aléatoire lors du choix des expériences :

- Si $\omega = 0$, les expériences vont être choisies de manière purement aléatoire
- Si $\omega = 1$, les seules expériences choisies vont être celles avec les valeurs de priorités les plus fortes

Toutefois, il est nécessaire ici de régler le problème d'overfitting : nous avons introduit un biais en faveur des expériences à haute priorité, qu'il n'y avait pas avant avec une sélection purement aléatoire.

Pour régler ce problème, nous introduisons également le concept d'Importance Sampling, en pondérant les expériences par le nombre de fois où elles sont déjà apparues.

## 6.2 Double Deep Q Learning

### 6.2.1 Théorie

Dans le cadre du Deep Q Learning, les valeurs "target" de la fonction Q sont prédites via la formule classique du Q Learning. Cette dernière utilise l'opérateur max sur les Q-values, or ces dernières sont bruitées, et ce d'autant plus au début de l'apprentissage. Par conséquent, des actions non-optimales peuvent avoir une meilleure Q-Value pour un état que l'action optimale, et ceci va au mieux ralentir, au pire détériorer l'apprentissage. Dans un papier de 2015 [3], Hado V.Hasselt propose une première solution pour résoudre le problème : il considère deux fonctions Q,en utilise une pour trouver l'action optimale (celle qui est l'argument maximal de la Q-Value) et l'autre pour mettre à jour la fonction Q. A chaque pas de temps, il alterne les deux fonctions. Dans notre projet, ce phénomène va être représenté par deux réseaux de neurones identiques qu'on notera $Q_{Target}$ et $Q_{Action}$. Notre méthode d'apprentissage va alors devenir la suivante :

- Un batch est récupéré depuis la mémoire d'expérience
- Le réseau $Q_{Action}$ est utilisé pour récupérer l'estimation des Q-values, et déterminer l'action optimale, c'est à dire celle l'argument maximal des Q-Values.
- Le réseau $Q_{Target}$ est cette fois-ci utilisé pour estimer les Q-values. Nous récupérons alors la Q-value associée à l'action choisie par le réseau $Q_{Action}$. De cette manière, le biais provoqué par le premier réseau ne sera pas le même que celui provoqué par le deuxième réseau.
- La Q-Target value est calculée, comme dans le cadre du Deep-Q-Learning.
- Les paramètres du réseau $Q_{Action}$ sont mis à jour.

Dans cette procédure, nous ne mettons à jour que les paramètres du réseau $Q_{Action}$. Pour mettre à jour les paramètres du réseau $Q_{Target}$, nous copions simplement les paramètres du réseau $Q_{Action}$ toutes les 10000 étapes. Ainsi, le réseau $Q_{Action}$ a toujours de l'avance sur le réseau $Q_{Target}$, mais ce dernier ne change que périodiquement, ce qui évite les problèmes de cible mobile.

### 6.2.2 Résultats

Voici une comparaison des performances sur le jeu Pong :

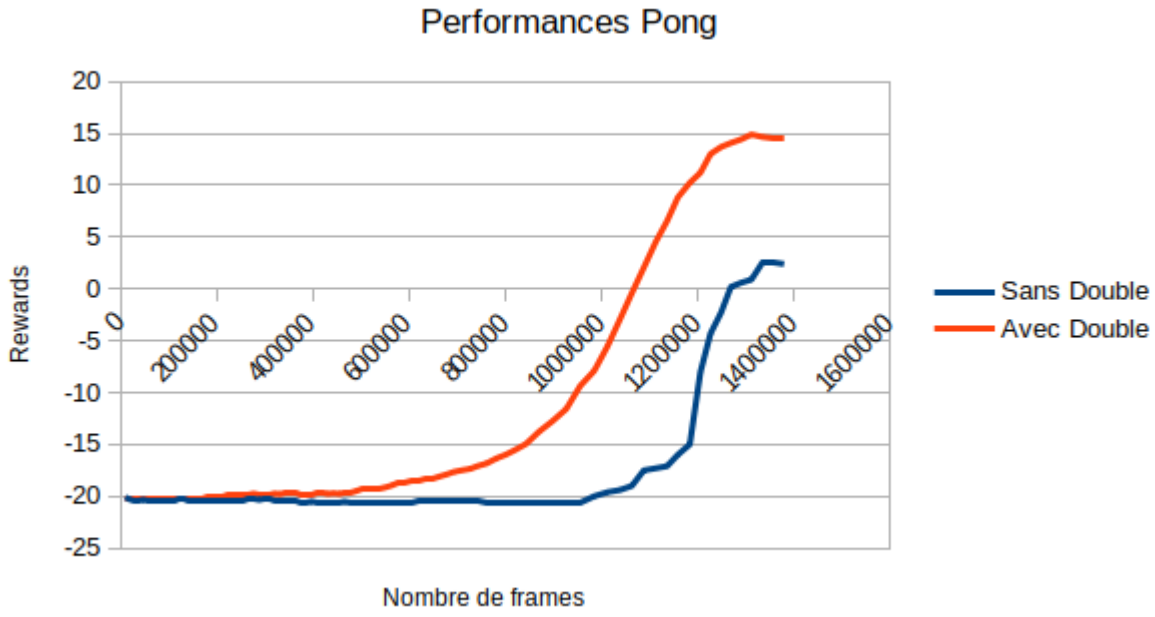



Figure 6.1: Comparaison Pong avec/sans Double Deep Q Learning

La courbe en bleu représente le Deep Q Learning classique, tandis que la courbe en rouge représente la version Prioritized Experience/ Double Deep Q Learning. L'axe des abscisses indique le nombre d'étapes réalisées tandis que l'axe des ordonnées représente la moyenne des 100 derniers scores. Nous pouvons alors observer que les performances avec double Deep-Q Learning sont bien supérieures. Notons que ces résultats ont été obtenus en environ une douzaine d'heures. Nous avons également comparé les performances sur le jeu Enduro :

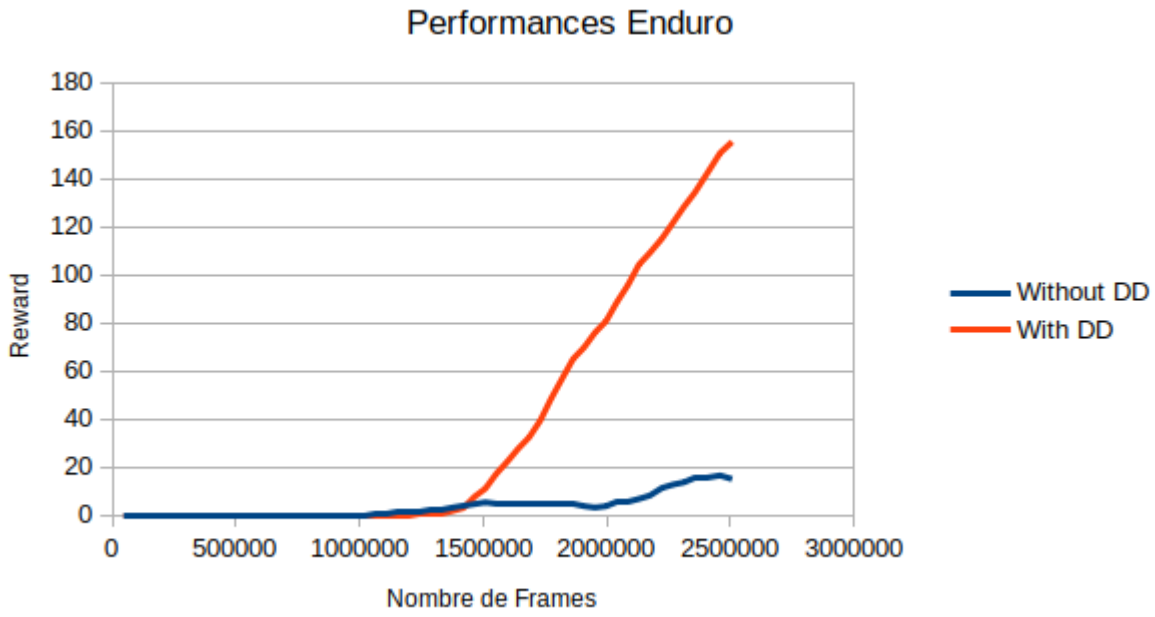



Figure 6.2: Comparaison Enduro avec/sans Double Deep Q Learning

Les résultats sont ici encore plus marqués : les deux méthodes obtiennent des récompenses nulles pendant un long moment, à cause de la nature du jeu, mais le Double Deep Q Learning arrive à décoller et apprendre de manière efficace au bout d'une dizaine d'heures, tandis que le Deep Q Learning classique stagne.

## 6.3 Dueling Network

### 6.3.1 Théorie

Dans le modèle du Deep Q Learning classique, nous souhaitons estimer les Q-Values, c'est à dire la récompense potentielle dans un état en appliquant une action définie. Toutefois, il est possible de décomposer cette valeur sous la forme $Q(s,a) = A(s,a) + V(s)$, avec :

- $V(s)$ : la valeur de l'état S
- $A(s,a)$ : L'avantage d'une action a dans un état s, c'est à dire si l'action a est meilleure que les autres actions dans cet état

Dans un papier de 2016 [4], Wang propose de calculer ces deux valeurs séparément. Pour expliquer l'intérêt de cette séparation, il prend l'exemple du jeu Enduro, présenté précédemment :

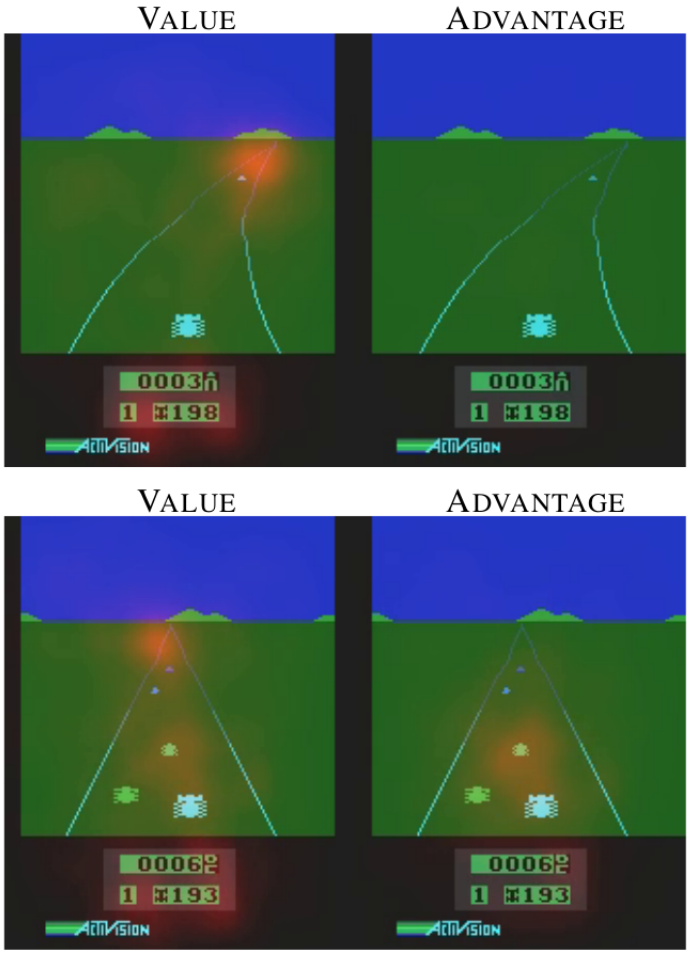



Figure 6.3: Valeur et Avantage

Wang explique alors que pour certains états, les actions n'ont pas d'impact sur l'environnement : par exemple, dans Enduro, lorsqu'il n'y a pas de voiture dans le champ de vision, il nous importe peu d'aller à gauche ou à droite. Cette séparation va ainsi nous permettre d'évaluer la valeur des états , sans avoir à apprendre l'effet de chaque action dans chaque état. Par conséquent, nous pourrons déterminer si un état est bon ou mauvais directement avec $V(s)$. Pour calculer ceci, nous allons devoir modifier légèrement notre réseau de neurones. Il va alors prendre la forme suivante :

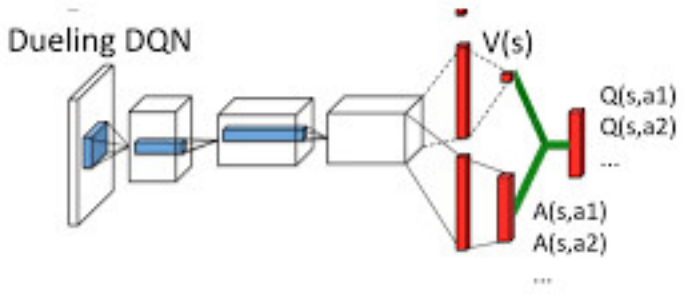



Figure 6.4: Dueling Network

En sortie de nos couches de convolution, nous allons séparer les valeurs en deux, et créer deux couches denses

- Une couche Value, qui admettra une valeur en sortie : $V(s)$
- Une couche Avantage, qui admettra $|A|$ valeurs en sortie : $A(a, s)$

Il nous reste alors à rassembler les deux sorties pour obtenir Q(s,a). Pour assurer la converge du modèle, il est nécessaire que $E(A) = 0$, c'est à dire que la moyenne des valeurs d'avantages soit égale à 0. Pour avoir ce résultat, nous posons alors la formule suivante :

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|A|} \sum_{a'} A(s,a'))$$

### 6.3.2 Résultats

Nous comparons tout d'abord les performances sur Pacman, pour obtenir les résultats suivants:

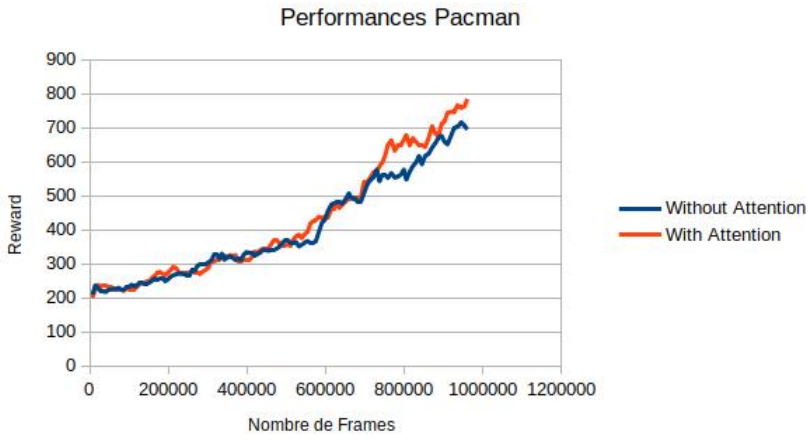



Figure 6.5: Comparaison Pacman avec/sans Dueling Network

L'amélioration n'est pas très perceptible, toutefois, Pacman ne semble pas être un jeu bien adapté pour ce concept, puisque peu importe l'état, nous avons envie de nous éloigner des fantômes , donc de choisir une action adaptée. Nous avons alors voulu réaliser des tests sur Enduro, puisque cela était le jeu choisi dans le papier. J'obtiens alors les résultats suivants :

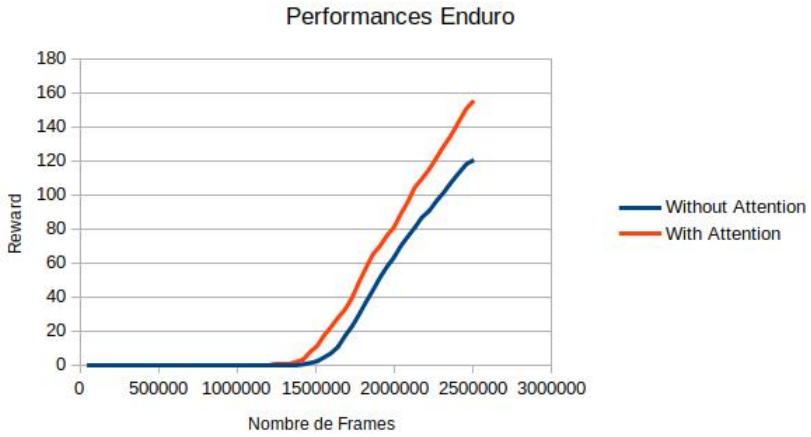



Figure 6.6: Comparaison Enduro avec/sans Dueling Network

Ici, les résultats sont plus perceptibles. Le réseau avec séparation des couches est plus performant.

# Chapter 7

# Conclusion

En conclusion , ce PFE m'aura permis de découvrir, conjointement avec mes cours de master l'apprentissage par renforcement. Cette méthode d'apprentissage, contrairement à l'apprentissage supervisé n'a pas besoin de données en entrée, mais simplement d'un modèle, représentant la description de son environnement (états), ce que l'agent à le droit de faire (actions), et l'impact de ses actes sur l'environnement (récompense).

L'apprentissage par renforcement est intéressant lorsqu'il n'y a pas une manière de réussir la tâche, mais plusieurs, c'est pourquoi il est notamment souvent utilisé pour les jeux : il n'y a pas une manière universelle de jouer à Pong ou Pacman. De plus, cet apprentissage est une méthode efficace, notamment pour trouver des petites failles ou manières de procéder auquel un humain n'aurait pas forcément pensé.

Par exemple, sur Pong, nous aurons pu voir que l'algorithme arrive à se placer de manière permanente dans une situation où il piège son adversaire, et enchaîne ainsi les points facilement et de manière identique.

La grande force de l'apprentissage par renforcement réside dans cette capacité à explorer de nombreuses possibilités, et ainsi pouvoir utiliser des stratégies inattendues. Par exemple, nous pouvons citer le coup 37 de la partie numéro 2 entre LeeSedol et AlphaGo : le coup joué par AlphaGo a complètement perturbé LeeSedol, car il semblait très faible, et n'aurait jamais été joué par un humain. Or ce coup s'est révélé décisif plus tard dans la partie. En apprenant à jouer seul, l'agent ne subit ainsi pas le biais humain : il n'aura donc pas d'à-priori sur certaines stratégies, ce qui peut être une force.

# Bibliography

[1] David Silver Volodymyr Mnih. “Playing Atari with Deep Reinforcement Learning”. In: (2013), p. 9.

[2] Tom Schaul. “PRIORITIZED EXPERIENCE REPLAY”. In: (2016), p. 21.

[3] Hado van Hasselt. “Deep Reinforcement Learning with Double Q-learning”. In: (2015), p. 13.

[4] Ziyu Wang. “Dueling Network Architectures for Deep Reinforcement Learning”. In: (2016), p. 15.