CentraleSupélec

**Projet : Reinforcement Learning**

CentraleSupélec

Saad Chtouki

# Codage d'un agent de Puissance 4

2023

*En groupe avec :*

Ibrahim Rakib - Guilhem Prince

# Table des matières

# 1 Introduction

Ce papier décrit l'élaboration d'un agent capable de jouer au puissance 4 à l'aide de l'environnement PettingZoo, dans le cadre du projet de Reinforcement Learning avec Guilhem Prince et Ibrahim Rakib, et ce en utilisant le Q-Learning. Le travail présenté peut-être vu comme la suite de celui présenté dans le rendu d'Ibrahim Rakib. En effet, nous avons en réalité travaillé conjointement les deux parties, chacun décidant de détailler une des deux parties dans son rapport. Certaines fonctions présentées ici ont par conséquent été écrites par Ibrahim (et vice-versa dans son rendu). Dans cette partie, nous allons développer une méthode plus élaborée pour entraîner notre agent à jouer de manière autonome. Nous allons nous baser sur la méthode de Q-Learning présentée par Ibrahim, mais en y détaillant plus spécifiquement l'état d'un joueur afin de tenter d'améliorer à terme les performances de l'agent (spécialement contre le `left_player` posant problème).

# 2 Implémentation

## 2.1 Rappel rapide de la méthode naïve

Pour un état $s_t$ et une action $a_t$, l'évolution de de la matrice de récompenses $Q(s_t, a_t)$ est la suivante :

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{ancienne valeur}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left( \underbrace{r_t}_{\text{récompense}} + \underbrace{\gamma}_{\text{facteur de réduction}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimation de la future valeur optimale}} \right)}^{\text{valeur apprise}}$$

L'implémentation précédemment présentée consiste à considérer l'état d'un joueur comme la dernière position `(x,y)` qu'il a joué, `x` et `y` étant respectivement dans $[0, 5]$ et $[0, 6]$. Cette idée d'implémentation, bien que performante contre un joueur aléatoire, n'est pas idéale puisque l'agent a une vision très limitée sur le déroulement de la partie. Le puissance 4 étant un jeu ou la stratégie à adopter dépend de la stratégie de l'adversaire (Comme la plupart des jeux à 2 joueurs d'ailleurs), il est nécessaire de prendre en compte les coups de l'adversaire dans l'état de l'agent.

## 2.2 Méthode utilisée

La représentation de l'état choisie est la suivante :

L'état d'un joueur à un instant $t$ de la partie n'est plus uniquement le couple $(x, y)$ du coup joué à l'instant $t-2$ mais le tuple :

$$((x, y), (a, b))$$

où $(x, y)$ sont les coordonnées coup joué à l'instant $t-2$ et $(a, b)$ sont les coordonnées coup joué **par l'adversaire** à l'instant $t-1$.

Ainsi la méthode décrite présente l'avantage de ne plus prendre uniquement en compte le dernier coup joué par l'agent mais à la fois son dernier coup et le dernier coup de l'adversaire, ce qui peut permettre une vision plus fine de la partie. La taille de matrice 5D `Q` est $(6, 7, 6, 7, 7)$ puisqu'un couple (état, action) est de la forme :

$$(x_{agent}, y_{agent}, x_{adversaire}, y_{adversaire}, action)$$

La matrice comporte en tout **12 348** valeurs.

Cette modélisation de l'état entraîne un déroulement différent de l'algorithme, notamment lors de la mise à jour des récompenses dans `Q`. En effet, dans l'approche naïve, le déroulement était le suivant :

1. Trouver la meilleure action `act` avec `get_actions()`.
2. Garder en mémoire l'état actuel dans la variable `last_state`.
3. Mettre à jour l'état au vu de l'action choisie.
4. Mettre à jour `Q[last_state, act] à l'aide de l'état actuel.`

Or, avec la modélisation décrite plus haut, on ne peut pas mettre à jour l'état une fois un coup joué, puisque l'état suivant dépend du coup que vas jouer l'adversaire. On ne peut donc pas mettre `Q` à jour à la fin de la boucle. C'est pour cela que l'on apporte deux modifications :

1. A chaque tour on mettra `Q` à jour en début de boucle. En effet, on récupérera la position jouée par l'adversaire afin de mettra à jour l'état et de calculer les nouvelles récompenses.
2. Si un coup est gagnant, on ne peut pas récupérer l'état suivant puisque ce dernier dépend du coup suivant de l'adversaire, qui n'arrivera pas si la partie est finie. Ainsi, en cas de victoire, la matrice `Q` est mise à jour de la façon suivante :

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot r_t$$

ou $a_t$ est l'action gagnante à partir de l'état $s_t$.

Notons que dans l'environnement PettingZoo, le reward `r` vaut 1 en cas de coup victorieux et 0 pour tout autre coup. C'est pour cela que lors de la mise à jour de Q en début de boucle, le reward r a été supprimé de la forumule car inutile (aura toujours pour valeur 0).

Une entité de la classe `Player` présente ainsi les attributs suivants :

1. Les attributs déjà présentés par Ibrahim, à savoir un **identifiant**, un **nom**, un boolean `left`, une matrice `Q` de taille $(6, 7, 6, 7, 7)$ et un "facteur d'exploration" `epsilon`.
2. Un état `last_st` et une action `last_action` indispensables pour garder en mémoire respectivement **l'action précédente** et **l'état précédent**.

La classe `Player` présente une méthode `get_actions()` adapté au type de joueur désiré :

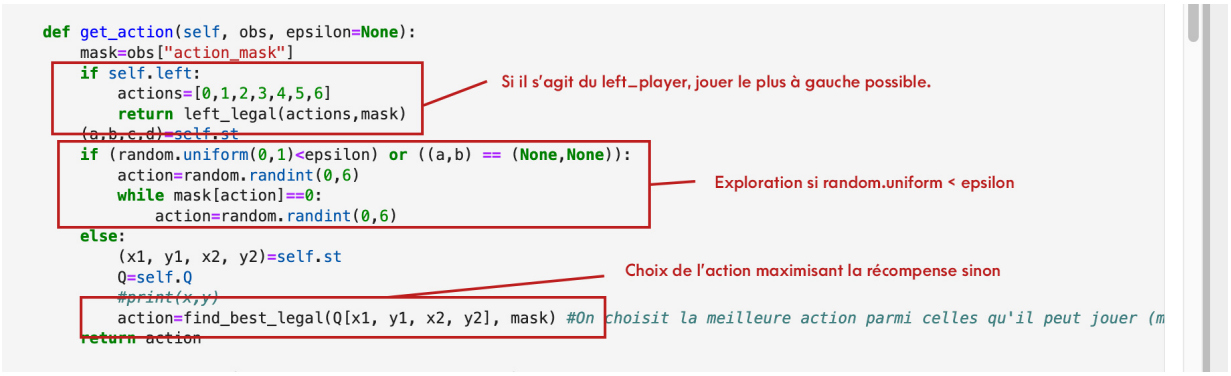


FIGURE 1 – Méthode `get_actions()`


FIGURE 2 – Fonction `play_game()`

## 2.3 Résultats

Comme pour l'approche dite naïve, Il a fallu faire un choix sur le partenaire d'entraînement de notre agent. Comme précisé par ibrahim, nous avons remarqué avec Saad que les résultats sont légèrement meilleurs lorsque l'agent affronte un joueur random qui n'apprend pas lors de la phase d'entraînement, ce choix est donc réalisé.

### 2.3.1 Entraînement

La matrice `Q` étant 42 fois plus grande que l'approche naïve, l'entraînement donc par conséquent être plus long. Il consiste ainsi en **100.000** parties contre un joueur random avec un facteur d'exploration `epsilon`=**0.3** (qui nous somble être un facteur performant parmi ceux essayés). Les paramètres sont les suivants :

1. `alpha`=0.1
2. `gamma`=0.9

Le choix d'alpha et gamma sera justifié plus bas. Voici par exemple un apperçu des valeurs de la

matrice multi-dimensionelle `Q` après 100 000 parties pour l'état (2,2,3,5) (ie si on a joué le coup précédent en (2,2) et notre adversaire a joué en (3,5) :

$$[0.46, 0.39, 0.71, 0.53, 0.39, 0.45, 0.40]$$

Ainsi, lorsque l'agent est dans l'état cité, la meilleure chose à faire est de jouer **l'action 2** (3ème colonne à partie de la gauche) puisque c'est cette action qui maximise la récompense.

### 2.3.2 Scores de l'agent

L'évaluation de l'agent consiste aussi en **40.000 parties**. On relève le nombre de victoires de l'agent lors de ces dernières. L'agent est mis en mode "exploitation" (à l'inverse d'exploration). En d'autres termes, on force son facteur d'exploration à zéro avant l'évaluation afin qu'il choisisse les actions maximisant les récompenses 100% du temps. Voici les résultats :

| Adversaire affronté | Nb de victoires | Nb de matchs nuls | Pourcentage de victoires |
|---|---|---|---|
| `Random_player` | 28845 | 40 | 72,2% |
| `left_player` | 14204 | 0 | 35,5% |

L'agent est donc performant contre un joueur aléatoire, mais étonnement un peu moins que l'agent implémentée avec la méthode naïve. Peut-être la matrice Q est-elle trop grande pour que l'entraînement soit assez pertinent (pour dépasser le score de la méthode naïve) ce qui est étonnant puisque l'agent codé à un point de vue très limité sur la partie en cours (connaissance uniquement du dernier coup joué). Ce qui est encore plus étonnant, c'est que l'agent est moins performant que l'agent naïf lorsqu'il affronte le `left_player`, en se faisant dominer 65% du temps. Autre phénomène inattendu, l'agent est mois performant contre `left_player` lorsqu'il s'entraîne justement contre ce dernier. Il est meilleur contre le `left_player` lorsqu'il s'entraîne contre le joueur aléatoire.

### 2.3.3 Choix des paramètres

Une recherche de grille fut implémentée afin de déterminer les paramètres `alpha` et `gamma` optimaux. On affiche dans le tableau les pourcentages de victoire contre un adversaire aléatoire sur **40000 parties** après entraînement :

| | `alpha` = 0.001 | `alpha` = 0.01 | `alpha` = 0.1 |
|---|---|---|---|
| `gamma` = 0.5 | 62.9% | 63.1% | 71.6% |
| `gamma` = 0.9 | 59.9% | 61.5% | 72.1% |

PERFORMANCE DES DIFFÉRENTS COUPLES DE PARAMETRES

Nous avons ainsi choisi `alpha`=0.1 et `gamma`=0.9. Notons qu'il est parfaitement logique qu'un taux d'apprentissage inférieur (inférieur au learning rate du cas naïf) soit plus performant pour le modèle actuel. La matrice`Q` étant ici bien plus importante, la **propagation des récompenses** doit se faire plus vite. En somme, le modèle a besoin d'apprendre plus vite au vu de la taille de `Q`.

# 3 Conclusion

Nous avons implémenté, en nous basant sur la méthode naïve présentée dans le compte rendu d'Ibrahim, une méthode d'apprentissage à priori plus élaborée, puisque l'on a tenté d'inclure la stratégie adverse dans la connaissance de l'état. Cette méthode n'a néanmoins pas porté ses fruits en terme de résultats puisqu'elle n'améliore pas les performances contre un joueur aléatoire (voire faire légèrement moins bien) ni celles contre le `left_player`, qui domine systématiquement notre agent (alors qu'il faisait en moyenne égalité avec l'agent naïf). On peut émettre des hypothèses sur l'absence de résultats satisfaisants, comme par exemple la taille de la matrice Q trop importante pour un apprentissage efficace, ou alors un nombre de runs pas assez suffisant. Une piste d'amélioration de la méthode de Q-Learning en général serait d'utiliser du Deep Q-Learning. Des réseaux de neurones possèdent l'architecture parfaite pour une tache comme une partie de puissance 4.