CentraleSupélec

Projet : Reinforcement Learning

CentraleSupélec

Ibrahim RAKIB

Codage d'un agent de Puissance 4

2023

En groupe avec :

Saad Chtouki - Guilhem Prince

Table des matières

1 Introduction

Ce projet d'apprentissage par renforcement (Reinforcement Learning) consistait en la conception d'un agent capable de jouer au Puissance 4, et ce, par deux méthodes. Nous avons conjointement, avec Saad, essayé un apprentissage par Q-Learning, qui vient compléter le contrôleur Monte Carlo élaboré par Guilhem.

Le Q-Learning est une technique d'apprentissage par renforcement qui permet à un agent d'apprendre à prendre des décisions en interagissant avec un environnement, et ce grâce à une table de gains Q, informant des récompenses de chaque action possible à partir de l'état de l'agent.

Mon approche se base donc sur une méthode naïve de Q-Learning, qui consiste à construire une table de valeurs Q n'offrant pas une vision complète du jeu pour l'agent. Nous verrons que cette méthode peut néanmoins présenter des résultats surprenants.

Dans ce papier, les étapes de l'implémentation seront décrites, en exploitant l'environnement `PettingZoo`. Les performances de l'agent confronté aux adversaires fournis dans le fichier fourni `project-starter.ipynb` seront par la suite évaluées.

2 Implémentation

2.1 Méthode utilisée

Le but du Q-Learning est de trouver une matrice multi-dimensionelle Q, qui assigne chaque couple (état de l'agent, action possible) à une récompense, traduisant l'intérêt d'une telle action. Pour un état s_t et une action a_t, l'évolution de de $Q(s_t, a_t)$ est la suivante :

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

La difficulté de l'implémentation réside ainsi dans le choix de l'état d'un joueur à un moment donné au jeu du puissance 4. Si on considère toutes les situations possibles, le jeu comptant 42 cases, 7 actions étant possibles et chaque case admettant trois possibilités (vide, jeton du joueur 1 ou jeton du joueur 2), il faudrait que la matrice Q soit de taille $7 \cdot 3^{42}$, ce qui semble peu raisonnable en termes d'espace mémoire.

Nous avons alors conjointement décidé avec Saad d'opter pour une méthode dite naïve, sur laquelle nous nous baserons pour élaborer une méthode plus complexe (cf. Rendu de Saad CHTOUKI), qui consiste à considérer l'état d'un joueur comme la dernière position `(x,y)` qu'il a joué (avec $(\mathtt{x},\mathtt{y}) \in [|0,5|] \times [|0,6|]$).

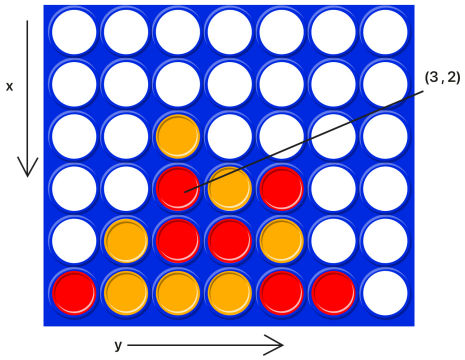

FIGURE 1 – Etat : Position x,y du dernier coup joué

Cette idée d'implémentation est bien entendu limitée, puisque l'agent n'a pas une vision complète du déroulement de la partie (il n'a en fait pas même connaissance des coups de son adversaire) mais agit uniquement en fonction de son dernier coup. On peut parler d'agent à l'aveugle. L'objectif de cette partie est de vérifier si ce type d'agent peut battre les agents fournis.

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

- Un identifiant `_id`, un nom `name` conçu à partir de l'`_id` et un booléen `left` indiquant si notre agent est un left_player.
- Une matrice `Q` de taille $(6, 7, 7)$.
- Un état `st` (couple (x,y)).
- Un "facteur d'exploration" `epsilon` correspondant à l'approche *Epsilon-Greedy* : il s'agit de la probabilité, pour l'agent, de choisir un coup au hasard à chaque tour. On parle ainsi de facteur d'exploration puisque l'agent, au lieu de choisir l'action maximisant la récompense, choisit un coup au hasard afin d'explorer certaines situations.
  Il est indispensable que `epsilon` soit strictement supérieur à 0 en phase d'entraînement. Sans cela, il jouera chaque partie de la même manière et tentera pas de nouveaux coups. Notons que si `epsilon` vaut 1, notre agent devient un agent aléatoire `Random`.

Nous avons co-écrit les fonctions relatives au fonctionnement de l'agent avec Saad (puisque celles-ci servent en majorité aussi pour sa partie). Voici une présentation de l'enchaînement de celles-ci :

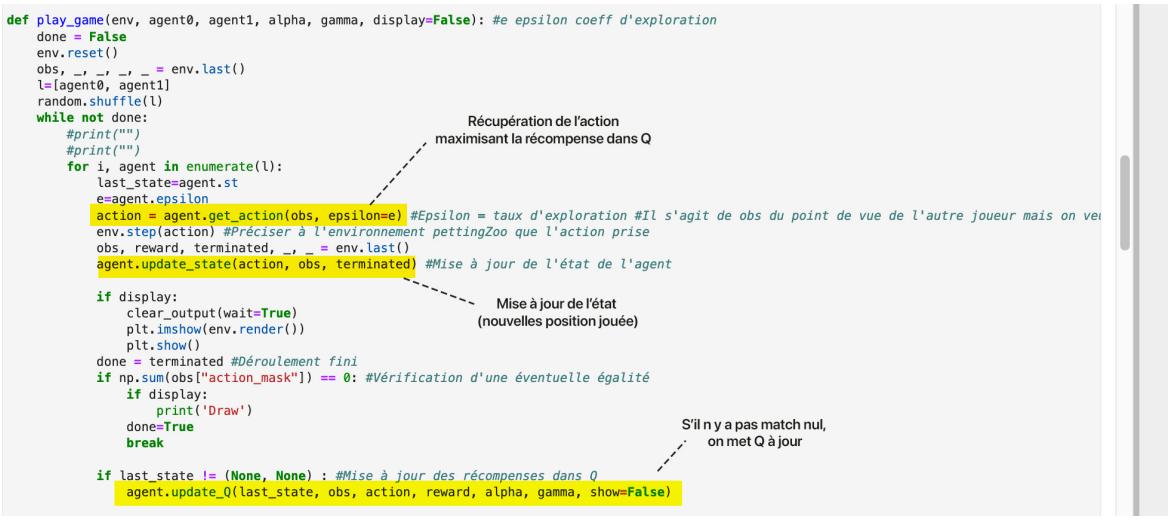

FIGURE 2 – Fonction `play_game()`

2.2 Résultats

Avant d'étudier les résultats, voici quelques remarques préliminaires que l'on peut noter afin d'expliquer certains choix d'implémentation :

1. Lorsque deux agents `Random` jouent l'un contre l'autre un grand nombre de fois, on s'aperçoit statistiquement que le joueur qui commence a environ 55% de chances de l'emporter (et non 50%). **Commencer présente donc un avantage**, c'est pourquoi dans la fonction `play_game()`, le joueur qui commence est tiré au sort.
2. Il a fallu faire un choix sur le partenaire d'entraînement de notre agent (un `Random`, un `left_player` ou un agent similaire au nôtre qui apprend également). 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, les entraînements sont donc ainsi réalisés contre un joueur `Random` (donc de paramètre `epsilon` valant 1). Cette meilleure performance peut être expliquée par la multitude de situations rencontrées en jouant contre un `Random`, et donc la richesse de la matrice Q.

2.2.1 Entraînement

L'entraînement de l'agent a donc consisté en **20.000** parties contre un joueur `Random` avec un facteur d'exploration `epsilon`=**0.3** (afin que l'agent puisse assez souvent mettre sa matrice Q à jour sans pour autant jouer tout le temps de la même manière puisque qu'il explore 30% du temps). Les paramètres sont les suivants :

1. `alpha`=0.01
2. `gamma`=0.5

Le choix des valeurs d'`alpha` et de `gamma` sera justifié plus bas. Voici par exemple un aperçu des valeurs de la matrice multi-dimensionelle Q pour l'état `(5,0)` (c'est à dire si on a joué en bas à gauche) :

[5,43e-3, 1.02e-2, 9.06e-3, 1.99e-2, 9.68e-2, **1.67e-1**, 1.33e-2]

FIGURE 3 – Récompenses pour chaque action lors de l'état (5,0), 20.000 parties d'entraînement

Ainsi, lorsque l'agent est dans l'état (5,0), la récompense de l'action 0 (jouer tout à gauche) est de $5,43e-3$. La meilleure action à jouer est l'action 5 (à savoir jouer en deuxième colonne en partant de la droite).

2.2.2 Scores de l'agent

L'évaluation de l'agent est également réalisée sur **20.000 parties**. On relève le nombre de victoires de l'agent lors de ces dernières. Voici les résultats :

Adversaire affronté	Nombre de victoires	Nombre de matchs nuls	Pourcentage de victoires
`Random_player`	17346	6	86,7%
`left_player`	10057	7	50,2%

FIGURE 4 – Evaluation de l'agent sur 20.000 parties, contre différents adversaires

L'agent est donc performant contre un joueur aléatoire, 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). Il est néanmois moins performant contre le `left_player`, même s'il s'entraîne contre ce dernier. Cela peut s'expliquer par le fait que la stratégie du `left_player` est facile à contrer si on a accès à ses coups, ce qui n'est pas le cas de notre agent, qui prend uniquement en compte son propre coup le plus récent.

2.2.3 Choix des paramètres

Une recherche de grille (`grid-search`) 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 **10 000 parties** après entraînement :

	`alpha` = 0.001	`alpha` = 0.01	`alpha` = 0.1
`gamma` = 0.5	61.5%	76.8%	76.5%
`gamma` = 0.9	59.3%	70.7%	76.8%

FIGURE 5 – Extrait de la `grid-search` pour quelques valeurs de nos paramètres `alpha` et `gamma`

Nous avons ainsi choisi `alpha`=0.01 et `gamma`=0.5. Notons que lors de l'évaluation sur 20.000 parties, l'agent a été encore plus performant (86.7% de victoires contre le joueur aléatoire comme évoqué plus haut).

3 Conclusion

La méthode naïve implémentée, bien qu'à priori sans intérêt puisque trop limitée, se révèle performante contre un joueur aléatoire (victoire dans 86.7% des cas lors de l'évaluation) mais demeure limitée contre le `left_player`, probablement car le jeu de ce dernier est redoutable lorsque son adversaire n'a pas connaissance de ses coups. Ces résultats sont à opposés à ceux de l'agent contrôlé par méthode de Monte Carlo implémenté par Guilhem, performant contre le `left_player` mais pas contre le joueur aléatoire. Dans le rendu de Saad figure une piste d'amélioration de la méthode présentée ici, où l'état ne prendra plus en compte uniquement les propres coups de l'agent mais également ceux de son adversaire.