CentraleSupélec

Projet Reinforcement Learning « Connect 4 »

Création d'Agents - Jeu du Puissance 4

Auteurs :
Mathian Akanati
Martin Bridoux
Faustine Malatray
Côme Stephant

Encadrant :
Hédi Hadiji
Céline Hudelot

Avril 2023

CentraleSupélec

Table des matières

1 Introduction et contexte

Le Reinforcement Learning est une technique qui permet à un agent de prendre des décisions dans un environnement complexe pour maximiser une récompense donnée. Lors de ce projet, nous nous sommes intéressés au Puissance 4 qui est un terrain de jeu idéal pour tester les performances d'un tel agent. Le jeu est simple en apparence, mais les possibilités sont nombreuses. Cela offre un environnement propice à la mise en place et à l'évaluation d'un agent d'apprentissage par renforcement.

Je me propose de détailler ma contribution au projet, en portant une attention particulière à la construction de l'agent basé sur la méthode de Q-Learning.

FIGURE 1 – L'homme rivalisera-t-il avec l'agent q-learner ?

2 Définition de l'agent Q-Learner

Le Q-Learning est une méthode d'apprentissage par renforcement qui s'adapte bien aux problèmes de décision séquentielle tels que le jeu de Puissance 4. En effet, le Q-Learning permet à un agent de prendre des décisions en fonction des états de l'environnement et des récompenses potentielles associées à ces décisions, ce qui est exactement le type de situation rencontrée dans le jeu de Puissance 4.

Tous nos programmes ont été conçus en Python. Nous utilisons des Jupyter Notebook, des fichiers Python qui contiennent nos différents agents et un fichier utils.py qui répertorie l'ensemble des fonctions d'intérêt. Dans mon cas, l'agent Q-Learner a été implémenté en dans le fichier `agents/q_learner.py`.

2.1 Hyperparamètres de l'agent

L'agent est initialisé avec plusieurs hyperparamètres :

- `alpha` : le taux d'apprentissage, qui détermine dans quelle mesure les nouvelles informations remplacent les anciennes connaissances ;
- `epsilon` : le taux d'exploration, qui contrôle la probabilité qu'un agent choisit une action aléatoire plutôt que l'action optimale en fonction de ses connaissances actuelles ;
- `gamma` : le taux d'escompte, qui détermine l'importance relative des récompenses immédiates et futures ;
- `epsilon_min` : la valeur minimale que peut prendre epsilon, qui permet de réduire progressivement la phase d'exploration de l'agent au fil du temps ;
- `epsilon_step` : la valeur qui indique le pas de la décroissance de epsilon.

L'agent est également doté d'une table de valeurs Q, `q_table`, qui stocke les valeurs Q pour chaque état d'observation et action possibles. Cette table est initialisée à `0` et mise à jour à chaque itération de la fonction d'apprentissage.

2.2 Méthodes de l'agent

Pour prendre une décision sur l'action à effectuer, l'agent utilise la méthode `get_action()`, qui suit une politique de décision ϵ-greedy : l'agent choisit l'action optimale avec une probabilité $1 - \epsilon$ (avec la méthode `best_choice_with_mask()`) et une action aléatoire avec une probabilité ϵ (avec la méthode `random_choice_with_mask()`).

Enfin, l'agent utilise la méthode `update()` pour mettre à jour la table de valeurs Q à chaque étape. Cette méthode prend en entrée l'état actuel de l'environnement, l'action choisie, la récompense associée, l'état suivant de l'environnement, ainsi que le booléen *done* qui indique si le jeu est terminé ou non. Elle met à jour la table de valeurs Q en fonction de la récompense obtenue et des valeurs Q associées à l'état suivant. De plus, elle réduit progressivement la valeur de epsilon à chaque étape en utilisant la méthode `epsilon_decay()`, qui réduit la valeur de epsilon selon un pas fixé. C'est cette méthode `update()` qui sera utilisée dans le processus d'apprentissage, afin de tenir compte des coups effectués.

3 Entraînement de l'agent

L'entraînement d'un agent utilisant le Q-Learning est primordial pour obtenir de bonnes performances, sans quoi l'agent aurait le même comportement qu'un agent aléatoire.

3.1 Apprentissage de la Q-table

J'ai mis au point la fonction d'apprentissage `train()` spécifique pour l'agent Q-Learner (et tout agent définit avec une méthode `update()` similaire). Cette fonction simule un grand nombre de parties contre un opposant, et à chaque coup met à jour la Q table de notre agent grâce à la méthode `update()`. À la fin de l'entraînement, un graphique montrant l'évolution du taux de victoire est affiché par la fonction.

Notre agent Q-Learner utilise une fonction de récompense qui donne une récompense de 1 lorsque l'agent gagne la partie, une récompense de -1 lorsqu'il perd la partie, et une récompense de 0 lorsqu'il y a match nul. La Q-table est mise à jour grâce à l'équation de Bellman, et en tenant compte de cette récompense :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1}$$

où s est l'état actuel, a est l'action choisie, r est la récompense reçue, s' est l'état suivant, α est le taux d'apprentissage, et γ est le facteur d'escompte.

3.2 Déroulé de l'entraînement

Afin de réduire le risque d'*overfitting*, il fallait réaliser la phase d'entraînement en faisant varier le plus de paramètres possible. Nous avons donc fait le choix de l'entraîner avec plusieurs agents différents. J'ai également ajouté des paramètres à la fonction d'entraînement :

- `first_player` afin de choisir si notre agent ou l'opposant commence la partie ;
- `alternate_first_player` pour alterner entre les deux au cours d'une même phase d'entraînement.

Cette fonction nous offre aussi le contrôle sur le nombre d'épisodes et le nombre de parties par épisode pour l'entraînement.

Agents ayant contribué à l'entraînement :

— RandomPlayer
— PlayLeftmostLegal
— MalynxDeep : Calcul avec une profondeur de 1
— MalynxAvoidingBlunder (MAB) : Détecte les coup où l'agent se ferait battre au prochain coup et évite de les jouer

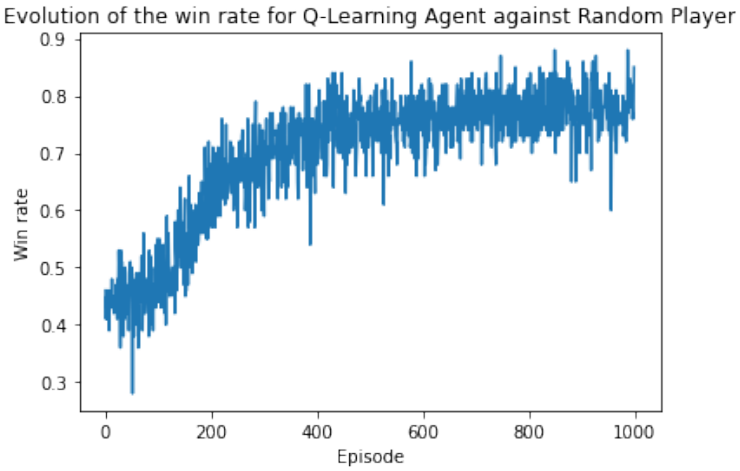

FIGURE 2 – Entraînement du Q-Learner contre un agent aléatoire

On peut voir sur ce graphique un point d'inflexion dans les performances de notre agent à l'épisode 200 de l'entraînement contre un agent aléatoire, signe d'un apprentissage.

4 Résultats et performances

L'intégralité du code permettant d'aboutir à ces résultats se trouve sur ce dépôt Github. Le Jupyter Notebook « project_starter »fournit tous les éléments pour retrouver ces résultats.

Observons les résultats de ces différents agents. Le pourcentage est calculé suivant ces règles :

— 1 point pour une victoire

— 0 point pour une défaite

— 0.5 point pour une partie nulle

Ainsi, si un agent réalise 2 victoires, 2 défaites et 3 parties nulles contre un autre agent, on considère que son taux de victoire est de 50%

Taux de victoire du Q-Learner entraîné, sur 1000 parties :

Opponent	Q-Learner
RandomPlayer starts	82%
RandomPlayer responds	90%
LeftPlayer starts	100%
Left Player responds	100%
MAB responds	100%
MalynxDeep responds	100%

Notre Q-Learner présente des performances remarquables contre les deux agents qu'on cherchait à battre, RandomPlayer et PlayLeftmostLegal. On note aussi qu'il performe mieux lorsqu'il commence la partie, ce qui est intuitif et peut être expliqué grâce à notre approche : lorque l'agent commence, l'état est toujours le même et donc l'agent choisit avec une probabilité $1 - \epsilon$ le départ optimal, qui est de commencer au milieu (appris au bout de quelques épisodes seulement). Lorsque l'agent joue en deuxième, il y a 7 états possibles et donc apprendre le coup optimal pour chacun de ces états demande une phase d'entraînement plus longue.

5 Conclusion

Dans ce projet, nous avons présenté notre implémentation d'un Q-Learner pour le jeu Connect4. Notre agent a été capable d'apprendre efficacement à jouer contre des adversaires de différents niveaux, en utilisant la mise à jour de sa Q-table basée sur l'équation de Bellman.

Nous avons omis de notre étude l'entraînement contre MalynxDeep et MAB lorsque notre agent commence en deuxième, car l'apprentissage prenait trop de temps sans améliorer les performances sur ces agents (et diminuant les performances sur les autres). Nous avons pensé à une approche de Deep Learning pour gérer un espace d'états aussi grand et ainsi rendre ce mécanisme d'apprentissage plus efficace, mais avons décidé de ne pas continuer sur cette piste au vu de sa complexité.