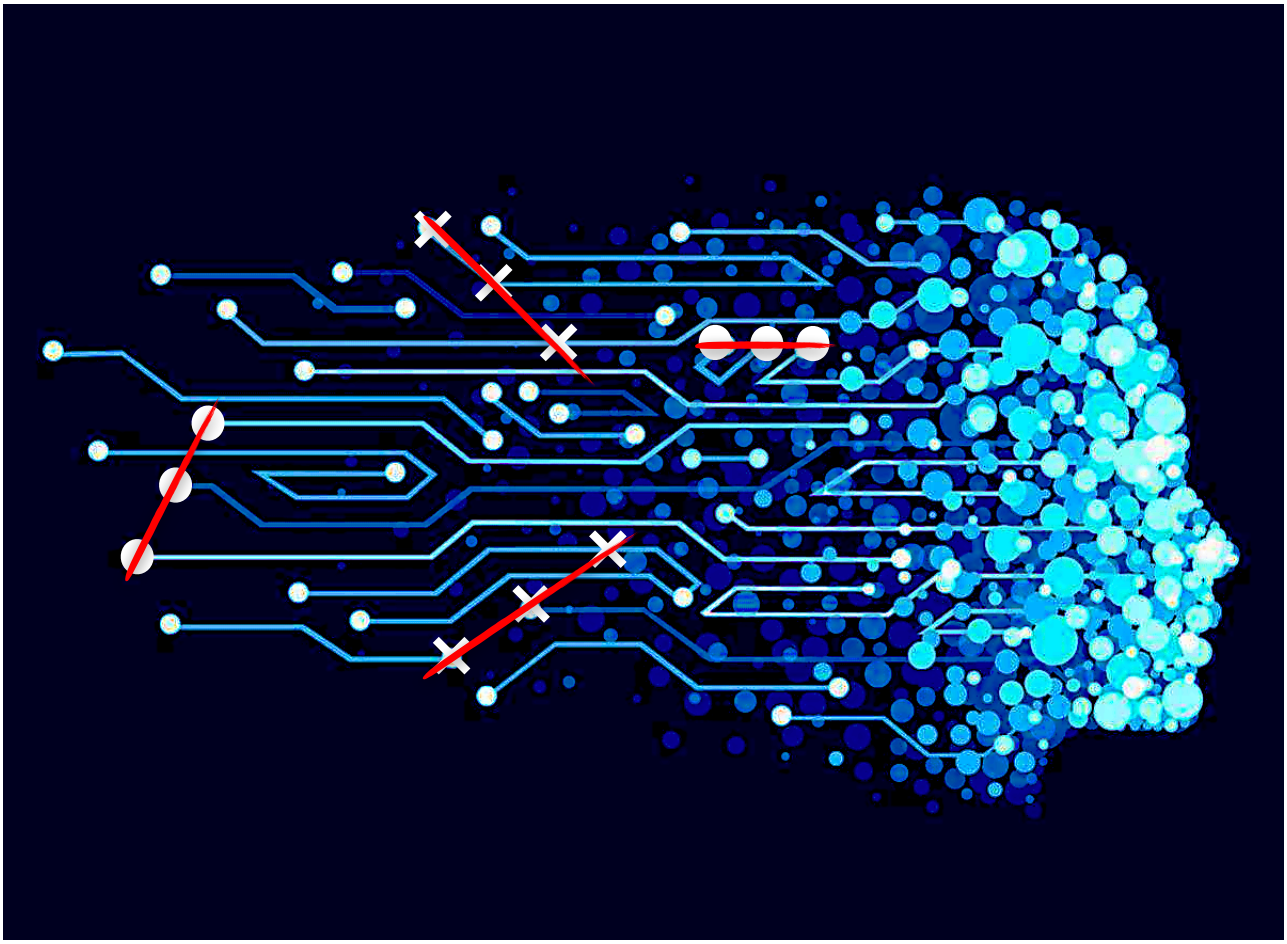
TIC TAC TOE

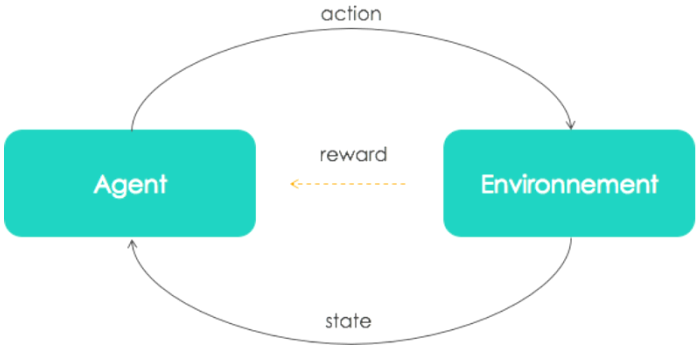
M2MO - REINFORCEMENT LEARNING

Salomé Amar - 19 avril 2020



INTRODUCTION

Les méthodes d'apprentissage par renforcement ont pris une place fondamentale dans les travaux de recherche en intelligence artificielle. Leur utilisation a été élargie aux jeux de société. Nous allons nous attacher à coder le célèbre jeu de réflexion Tic Tac Toe par apprentissage par renforcement. Tout d'abord, nous allons entrainer deux agents à jouer l'un contre l'autre et sauvegarder leur politique, puis nous chargerons cette politique et l'agent jouera contre un humain.



CHOIX DES MÉTHODES

Le code présenté est scalable pour toute grille de jeu I x J x K mais nous nous intéressons ici au cas I = J = K = 3. Comme pour tout problème d'apprentissage par renforcement, nous commençons par la définition précise des trois paramètres suivants :

- Les états s : on initialise une grille 3 x 3 remplie de 0 indiquant les positions disponibles qui sera mise à jour de 1 pour les X (par défaut le joueur 1) et de -1 pour les O (par défaut le joueur 2). En hashant cette grille, on obtient une table, par exemple {1, -1, -1, -1, 1, 1, 0, 0, 0}. Dans notre cas, il y a 3^{3^2} = 19 683 états dont un bon nombre seront impossibles.

- Les actions a : si la partie n'est pas terminée, elles consistent en le remplacement d'un 0 en -1 ou 1. Une action est spécifique à un état. Dans notre cas, il y a 19 683 × 9 = 177 147 paires états-actions, mais toutes ne sont pas possibles.

- Les récompenses r : à chaque état comportant trois X ou O alignés verticalement, horizontalement ou diagonalement est assigné une récompense de 1, et tous les autres une récompense de 0. Les récompenses ne sont données qu'à la fin de la partie.

Puisqu'on peut difficilement prédire l'état de l'environnement au pas de temps suivant, notre modèle d'environnement sera *model free*. L'agent s'intéressera aux récompenses associées aux paires états-actions et n'aura pas à explorer tout l'environnement.

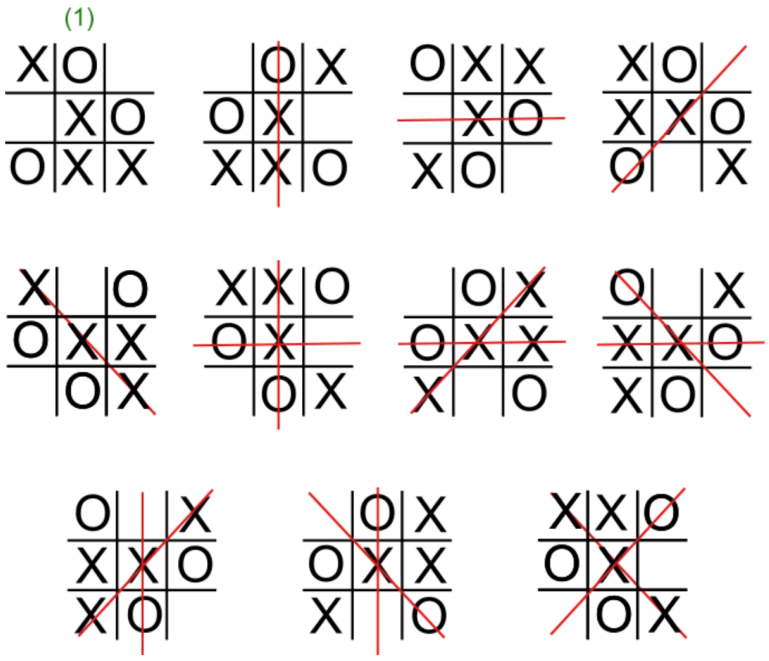
On fait l'escompte des récompenses seulement à la fin de la partie, on raisonne donc en terme de retours. L'agent doit choisir une méthode On-policy, il n'explorera pas les états passés non explorés. A la fin de la partie, toutes les paires (s,a) visitées et leur retour associé servent à actualiser Q(s,a).
Pour mettre à jour l'estimation de la valeur des états, nous choisissons d'appliquer l'algorithme SARSA :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \gamma \left[r_{t+1} + \alpha Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Ce dernier a pour objectif de déterminer une politique optimale, qui permettra à l'agent de choisir les actions qui le mèneront à la victoire : l'agent pourra alors choisir la meilleure action quel que soit l'état dans lequel il se trouve.

En termes de choix d'actions, nous utiliserons une méthode ϵ-greedy pour équilibrer l'exploration et l'exploitation, qui est simple à implémenter et plus rapide que Softmax puisqu'elle ne nécessite pas d'obtenir une distribution pour chaque état lors de l'exploration.

L'effet d'invariance par rotation et symétrie, qui est propre au jeu, nous permettra de visiter de nombreuses paires états-actions supplémentaires sans exploration et donc d'augmenter la vitesse de convergence.



Effets de symétrie et de rotation pour
une grille de jeu de morpion 3 x 3

PRÉSENTATION DU CODE

Le programme s'articule autour de trois classes : State, Player et HumanPlayer. Pour commencer, nous créons une classe State qui permet d'enregistrer l'état de la grille de jeu, de la mettre à jour lorsqu'un joueur joue, de l'afficher, de décréter si la partie est finie et de récompenser les joueurs en conséquence.

Les attributs de la classe State sont :

- **p1** et **p2** : deux joueurs. Par défaut, p1 est X et p2 est O.
- **board** : grille de jeu remplie de 0 (ici de dimension 3x3).
- **playerSymbol** : matérialise le tour d'un joueur, vaut 1 lorsque c'est à p1 de jouer et -1 lorsque c'est à p2. Par défaut, p1 vaut 1. Dès qu'un joueur effectuera une action, playerSymbol changera de valeur par la méthode updateState.
- **isEnd** : booléen qui permet de savoir si le jeu est fini ou non (False par défaut). On rappelle que le jeu est terminé si l'un des deux agents a aligné (verticalement, horizontalement ou diagonalement) trois fois son symbole ou s'il n'y a plus de positions disponibles sur la grille de jeu.

- **boardHash** : None par défaut, il prendra comme valeur la grille de jeu hachée par la méthode getHash.

Les méthodes de la classe State sont :

- **getHash** : hache la grille de jeu.
- **showBoard** : permet d'afficher l'interface du jeu.
- **availablePosition** : garde en mémoire la liste des cases vides à chaque tour.
- **updateState** : lorsqu'un joueur effectue une action, permet de mettre à jour la grille en ajoutant le symbole correspondant et de changer le tour à l'autre joueur (par changement de valeur de l'attribut playerSymbol).
- **winner** : permet de vérifier si trois symboles sont alignés (verticalement, horizontalement, diagonalement) auquel cas un joueur a gagné (renvoie 1, -1 respectivement pour p1, p2), si le jeu est à égalité (renvoie 0) ou si le jeu n'est pas fini (renvoie None). Cette méthode met également à jour la valeur de l'attribut isEnd selon que le jeu soit fini ou non.
- **giveReward** : selon le résultat de la méthode winner(), on attribut à chaque joueur sa récompense en utilisant la méthode feedReward de la classe Player. Le gagnant se verra attribuer comme récompense 1, le perdant 0, et en cas d'égalité, chaque joueur aura 0.
- **reset** : réinitialise la grille de jeu (sera utilisée en fin de jeu pour recommencer une partie).
- **play**
- **play2**

La classe Player permettra à notre agent d'apprendre et d'affiner sa politique en jouant contre un autre agent. Les actions seront choisies selon l'estimation actuelle des états. Tous les états du jeu seront enregistrés en mettant à jour l'estimation de la valeur des états après chaque partie. Nous pourrons ensuite enregistrer la politique et la charger pour l'utiliser.

Les attributs de la classe Player sont :

- **name** : le nom du joueur, à définir par l'utilisateur.
- **states** : une liste contenant les états (vide par défaut).
- **states_values** : un dictionnaire reliant les états à leur valeur (vide par défaut).

- **epsilon** : la valeur de l'hyper-paramètre ϵ de la méthode ϵ-greedy. Par défaut, on prend ϵ = 0.3, ce qui signifie que 70% du temps, l'agent prendra une action basée sur l'estimation actuelle de la valeur des états (*exploitation*), et 30% du temps, l'agent prendra une action aléatoire (*exploration*).
- **learning_rate** : la vitesse d'apprentissage pour actualiser la politique, qu'on prend par défaut égal à 0.2.
- **alpha** : taux d'actualisation dans l'algorithme SARSA, qu'on prend par défaut égal à 0.9.

Les méthodes de la classe Player sont :

- **getHash** : pour obtenir le hachage de la grille de jeu.
- **chooseAction** : permet de choisir les actions parmi les actions possibles, en choisissant celle qui a la plus grande valeur, selon une méthode d'exploration/ exploitation ϵ-greedy.
- **addState** : ajoute l'état en paramètre à la liste des états.
- **feedReward** : appelée par la méthode feedReward de la classe State, permet de mettre à jour la valeur des états par l'algorithme SARSA, en fonction des récompenses.
- **reset** : réinitialise la liste d'états.
- **savePolicy** : permet de sauvegarder la politique du joueur.
- **loadPolicy** : permet de charger la politique du joueur.

Nous créons enfin une classe HumanPlayer qui permettra, une fois que l'agent aura appris à jouer en s'entrainant suffisamment avec un autre agent, de jouer contre un humain. Pour rappel, le joueur humain jouera toujours après l'agent.

Cette classe a également un attribut **name**, qui permet à l'utilisateur de rentrer un nom au joueur, et une méthode **chooseAction**, qui cette fois permet à l'utilisateur de rentrer l'action qu'il souhaite.

```
Entrez la colonne de jeu :0
Entrez la ligne de jeu :0
-------------
| o |   |   |
-------------
|   | x |   |
-------------
|   |   |   |
-------------
```

Maintenant que nos classes Player et HumanPlayer sont expliquées, nous pouvons décrire le comportement des deux dernières méthodes de la classe State :

- **play** : permet à l'agent d'apprendre sa politique en s'entrainant contre un autre agent, selon un nombre de jeux donné en argument (vaut 400 000 par défaut). A chaque jeu, on choisit aléatoirement celui qui commence à jouer. Tant que le jeu n'est pas terminé (condition testée par la valeur de l'attribut isEnd), p1 et p2 jouent à tour de rôle. On fait appel successivement aux méthodes availaiblePosition, chooseAction (de la classe Player), updateState, addState (de la classe Player). Ainsi, on choisi une position parmi les positions disponibles, on met à jour la grille de jeu en fonction de ce choix, on ajoute cet état à la liste des états, et également les états symétriques et rotatoires à celui-ci dans le cas où la grille de jeu est carrée (afin de garder en mémoire les effets d'invariance pour converger plus vite). On vérifie avec la méthode winner si il y a un gagnant. Si l'un des joueurs a gagné, on attribue les récompenses par la méthode giveReward (qui elle-même fait appel à la méthode feedReward de la classePlayer ; les valeurs sont actualisés par l'algorithme SARSA) et on réinitialise la liste d'états des deux joueurs. Dans le cas contraire, le tour est à l'autre joueur et on retrouve le même déroulement.

- **play2** : permet à l'agent de jouer contre un joueur humain. Par la valeur de l'argument begin, l'utilisateur peut décider de commencer à jouer ou non (par défaut, c'est l'agent qui commence). Lors du tour de l'agent, on choisi une position parmi les positions disponibles (méthode availablePosition et chooseAction de la classe Player), on met à jour la grille de jeu en fonction de ce choix (méthode updateState), on ajoute cet état à la liste des états (méthode addState de la classe Player). Lors du tour du joueur humain, le choix de l'action va appeler la méthode chooseAction de la classe HumanPlayer qui permet à l'utilisateur de rentrer les positions de son choix. On vérifie à chaque tour avec la méthode winner si il y a un gagnant et on suit le même déroulement que lorsque deux agents jouent.

Nous pouvons ainsi jouer. Nous créons deux agents, nous les faisons s'entrainer (de l'ordre de 400 000 tours pour obtenir une puissance surhumaine) et nous enregistrons leur politique. Ainsi, nous pouvons jouer nous-même contre un agent en chargeant sa politique.

POUR ALLER PLUS LOIN

Afin d'améliorer notre approche, les valeurs des hyper-paramètres peuvent être modifiées afin d'augmenter la vitesse de convergence. Pour trouver les meilleurs paramètres, nous pouvons faire différents tests ou utiliser des valeurs dont les résultats ont déjà démontré l'efficacité. Pour rappel, nous avions choisi : epsilon = 0.3, alpha = 0.2, et learning_rate = 0.9.

Nous pouvons également ajouter les effets d'invariance par symétrie et rotation dans le cas de où la grille de jeu est non carrée (typiquement $I \neq J$). Nous avons simplement ajouté, dans le cas où la grille de jeu est carrée, les effets d'invariance par rotation de 90 degrés (vers la gauche et vers la droite), rotation de 180 degrés, symétrie horizontale, symétrie verticale. Nous pouvons rajouter les symétries diagonales gauche et droite.

Nous pouvons également envisager de changer les récompenses. Par exemple, si un joueur commence et que la partie termine à égalité, l'autre joueur peut se voir attribuer une récompense de 0.3 (au lieu de 0 actuellement).