Introduction au Q-learning

Fabien Teytaud *

Université du Littoral Côte d'Opale

17 février 2021

Rappels rapides sur l'apprentissage par renforcement

L'apprentissage par renforcement est différent de l'apprentissage supervisé que nous avons vu précédemment car nous n'avons plus la possibilité d'avoir des informations détaillées correspondant au problème (caractéristiques, étiquettes).

Dans ce type d'apprentissage, nous avons un agent qui évolue dans un environnement, qui reçoit de ce dernier des informations qui lui permettent de prendre des décisions. Pour chaque décision, l'agent reçoit une récompense qui va lui permettre de renforcer ses choix. Le but de l'agent est de trouver une politique qui va maximiser l'espérance des gains des situations qu'il va rencontrer ou des actions qu'il va prendre.

Dans le TP précédent nous avons vu comment estimer cette politique à l'aide de l'algorithme UCB. Dans ce TP nous allons voir une autre méthode : l'algorithme du Q-learning.

*Electronic address: teytaud@univ-littoral.fr

Description du problème

Le problème est le suivant : un pirate cherche un trésor, pour cela il va se déplacer sur une île. L'île est représentée par une matrice de cette façon :

— Une case d'une valeur 0 ne représente rien de particulier.
— Une case d'une valeur 2 représente une bouteille de Rhum.
— La case d'une valeur 10 représente un trésor.

Si le pirate trouve un trésor, alors il quitte l'ïle riche et le jeu est terminé.

Créer cet environnement. On aimerait pouvoir changer quelques paramètres, comme la taille de l'île, et le nombre de bouteilles de Rhum. Les emplacements des bouteilles de Rhum et du trésor doivent être aléatoires.

L'algorithme ϵ-greedy

Dans un premier temps nous allons implémenter l'algorithme ϵ-greedy. L'algorithme est très simple. En partant d'une position initiale aléatoire, le déplacement se fait :

— en prenant l'action qui a le plus grand score parmi les actions possibles avec une probabilité 0.9 (vous pouvez jouer avec la valeur d'ϵ pour bien comprendre son intérêt),
— de façon aléatoire avec une probabilité 0.1 (ou $1 - \epsilon$).

Nous allons rajouter un critère : si le pirate n'a pas trouvé le trésor en moins de 20 déplacements, alors nous allons considérer qu'il ne le trouvera jamais (ou qu'il est trop saoul pour le trouver).

Il est conseillé pour cette partie de faire une fonction qui pour un état renvoie les actions possibles, ainsi qu'une fonction qui retourne une action aléatoire possible (cf Etape 1 du Q learning).

Résultats attendus

Nous voyons que cet algorithme a une très forte tendance à converger localement. Nous allons maintenant étudier l'algorithme du Q-learning afin de voir s'il se comporte mieux.

Le Q-learning

Le principe

Tout d'abord nous allons devoir nous familiariser avec deux notions :
— Etat
— Action

Un état est une position (une case dans notre exemple) dans laquelle l'agent (le pirate dans notre exemple) se situe. Une action est un mouvement de l'agent. Il peut aller (quand cela est possible) au Nord, au Sud, à l'Est ou à l'Ouest.

Nous pouvons définir une matrice état $\leftrightarrow$ action pour notre pirate. Par exemple, pour le monde suivant :

0	10
0	0
2	0
0	0

nous avons 8 états possible :

s_0	s_1
s_2	s_3
s_4	s_5
s_6	s_7

et nous définissons la matrice état $\leftrightarrow$ action suivante :

	N	S	E	O
s_0	-1	0	10	-1
s_1	-1	0	-1	0
s_2	0	2	0	-1
s_3	10	0	-1	0
s_4	0	0	0	-1
s_5	0	0	-1	2
s_6	2	-1	0	-1
s_7	0	-1	-1	0

Un *-1* represente une action impossible par rapport à l'état actuel, une autre valeur correspond à la récompense que l'on reçoit. Par exemple, de la position s_3, si le pirate va au Nord il gagne le trésor, s'il va au Sud il se retrouve en s_5, il ne peut pas aller à l'Est et s'il va à l'Ouest il se retrouve en s_2.

Dans l'algorithme sur Q-learning, notre agent va devoir apprendre une matrice similaire, et va représenter une sorte de mémoire de l'agent, une sorte de réprésentation qu'il a du monde qui l'entoure.

La règle de transition du Q-learning est très simple :

$$Q(s,a) = R(s,a) + \gamma * \max_{a'}(Q(s',a'))$$

avec s un état et a une action. $R(s,a)$ correspond à la matrice état $\leftrightarrow$ action précédente, pour un état et une action donnés.

Nous pouvons remarquer que la valeur de Q pour un couple état/action correspond à la somme de la valeur dans la matrice état $\leftrightarrow$ action et du paramètre d'apprentissage γ multiplié par le maximum de Q pour toutes les actions possibles à partir de l'état suivant.

L'idée est que l'agent va apprendre par l'expérience. Il va explorer les états jusqu'à trouver l'état final.

Exemple : application sur notre pirate

Nous allons maintenant créer cette matrice Q. Pour la suite nous fixons $\gamma = 0.9$.

Tout d'abord il nous faut initialiser la matrice à 0. Ensuite, nous allons faire une exploration de l'espace jusqu'à atteindre un état final (cette exploration s'appelle en général un épisode).

Voici notre matrice Q à l'état initial :

	N	S	E	O
s_0	0	0	0	0
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0
s_6	0	0	0	0
s_7	0	0	0	0

Supposons que l'état initial choisi soit s_0. Si nous regardons dans la matrice R nous voyons que 2 actions sont possibles : S et E. Disons qu'aléatoirement l'action E soit choisie. En choisissant E, l'état d'arrivée serait s_1. Si le pirate était dans cet état il aurait 2 possibilités : S et O.

Nous devons calculer :

$$\begin{aligned} Q(0,E) &= R(0,E) + \gamma * \max_{i \in S,O}(Q(1,i)) \\ &= 10 + 0.9 * \max(0,0) \\ &= 10. \end{aligned}$$

Le nouvel état devient donc s_1. Comme il s'agit de notre trésor, notre état terminal, cela signifie que ce premier épisode est terminé.

Voici notre matrice Q après ce premier épisode :

	N	S	E	O
s_0	0	0	10	0
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0
s_6	0	0	0	0
s_7	0	0	0	0

Nous commençons un nouvel épisode en choisissant alétoirement un état initial : s_2. A partir de cet état nous voyons qu'il a 3 actions possibles : N, S et E. Aléatoirement N est choisi. Nous sommes donc dans l'état s_0, qui a

deux actions possibles S et E. Comme précédemment nous calculons la valeur de Q : Nous devons calculer :

$$\begin{aligned} Q(2,N) &= R(2,N) + \gamma * \max_{i \in S,E}(Q(0,i)) \\ &= 0 + 0.9 * \max(0,10) \\ &= 9 \end{aligned}$$

La matrice Q devient donc :

	N	S	E	O
s_0	0	0	10	0
s_1	0	0	0	0
s_2	9	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0
s_6	0	0	0	0
s_7	0	0	0	0

L'état 0 n'est pas terminé donc l'épisode continue. Une action est choisie aléatoirement entre S et E. Disons que E est choisi. En choisissant E, l'état d'arrivée serait s_1 et il aurait 2 possibilités : S et O.

Nous devons calculer :

$$\begin{aligned} Q(0,E) &= R(0,E) + \gamma * \max_{i \in S,O}(Q(1,i)) \\ &= 10 + 0.9 * \max(0,0) \\ &= 10. \end{aligned}$$

Donc dans ce cas la valeur ne change pas. L'état s_1 étant terminal l'épisode s'arrête.

Ce processus est itéré un certain nombre d'épisodes, et on espère que l'agent apprenne une matrice Q cohérente.

Implémentation de l'algorithme

Nous allons faire cet algorithme pas à pas.

Etape 1

Tout d'abord rajouter à votre classe environnement une fonction qui pour un état renvoie les actions possibles, ainsi qu'une fonction qui retourne une action aléatoire possible.

Etape 2

Ensuite, rajouter une fonction à votre classe Environnement qui calcule la matrice *état* $\leftrightarrow$ *action*.

Etape 3

Ensuite, rajouter une fonction à votre classe Environnement qui à partir d'un état et d'une action renvoie un nouvel état.

Nous avons maintenant les éléments nécessaires pour démarrer notre apprentissage.

Etape 4

Créer une classe Agent. Un agent à un paramètre γ et une matrice Q initialisée à 0.

Etape 5

Dans la classe Agent ; écrire une fonction qui à partir d'un état s et d'une action a calcule $Q(s, a)$, qui met à jour la matrice et qui retourne le nouvel état.

Etape 6

Dans la classe Agent ; écrire une fonction qui effectue un épisode de l'algorithme du Q learning. Cette fonction choisit aléatoirement un état initial et une action initiale et appelle la fonction de la question précédente (sur le nouvel état et avec une action aléatoire) tant qu'un état final n'est pas rencontré.

Etape 7

Dans la classe Agent, écrire une fonction qui effectue n épisodes. Il s'agit d'une simple boucle sur la fonction de la question précédente.

Etape 8

C'est bientôt terminé. Nous avons appris notre matrice Q. Maintenant nous souhaitons l'utiliser. Pour cela, à partir d'un état initial, notre pirate va simplement choisir l'action max dans la matrice Q. Il devrait trouver le trésor rapidement grace à son apprentissage !

Modifier le paramètre γ afin de voir son impact dans l'apprentissage.

Etape finale

Question 1

Rajouter une fonction `random_step` qui prend en entrée uen position (x, y) et un entier `nb_steps_max`. Vous devez faire des actions aléatoires tant que le trésor n'est pas trouvé et que le nombre d'actions effectuées est inférieur à `nb_step_max`. La fonction doit retourner `nb_step_max` moins le nombre d'actions effectuées pour trouver le trésor (0 si le trésor n'a pas été trouvé).

Question 2

Rajouter une fonction `Monte_Carlo` qui pour chaque coup possible effectue un appelle à `random_step`. L'action finalement choisi par Monte-Carlo sera l'action qui aura le meilleur score. Boucler jusqu'à atteindre le trésor.

Question 3

Faire une fonction qui compare les 3 méthodes développées dans ce tp (ϵ-greedy, Q-learning et Monte-Carlo).

Exemple de résultats attendus :

```
10 xps, environment size 3x5, 10 local optima
        Q leaning: 3.9 steps (0.04155285358428955s per run)
        E greedy: 111.8 steps (0.0008421182632446289s per run)
        MonteCarlo: 6.8 steps (0.0010586023330688477s per run)

10 xps, environment size 5x7, 10 local optima
        Q leaning: 3.0 steps (0.12863826751708984s per run)
        E greedy: 335.6666666666667 steps (0.0020469029744466147s per run)
        MonteCarlo: 6.333333333333333 steps (0.0012052059173583984s per run)
```