Apprentissage par renforcement

1 Introduction

Un programme, un robot peuvent-ils apprendre à se comporter dans un environnement inconnu ? On sait qu'un chien peut être dressé par punition-récompense et que des animaux généralement jugés moins intelligents, comme les oies ou les corbeaux, sont capables de modifier durablement leur comportement en fonction d'essais et de réponse du monde extérieur. Ce type d'apprentissage, fondamental dans le monde animal, est-il modélisable et transposable pour des programmes ? Les techniques d'apprentissage par renforcement visent à faire cette transposition.

L'apprentissage par renforcement a de très nombreuses applications potentielles. Par exemple, dans le domaine financier, des algorithmes de trading; de gestion de portefeuille ou de pricing d'options, de robot jouant capable de jouer seul aux jeux vidéo utilisent des techniques d'apprentissage par renforcement. Les systèmes de recommandation que l'on trouve sur la toile (recommandation de produits, optimisation du placement des publicités, etc...) recourent aussi à l'apprentissage par renforcement. On peut aussi penser à l'optimisation des vélos en libre service, etc...

Toutes ces applications nécessitent des algorithmes capables de maximiser un gain dans un milieu mal connu, donc avec impossibilité d'une programmation a priori, et éventuellement sujet à évolutions.

2 Découverte par l'exemple

Comme ceci est illustré par la figure 1, un problème d'apprentissage par renforcement (AR) met en scène un agent en interaction avec un environnement. L'agent reçoit de son environnement un ensemble de perceptions qui déterminent son état. Il décide en retour d'une action qui modifie sa situation dans l'environnement, donc son nouvel état. En outre, l'agent reçoit de l'environnement des récompenses lorsqu'il est dans une bonne situation et des punitions lorsqu'il est dans une mauvaise situation. Pour un agent donné, l'objectif assigné par un problème d'apprentissage par renforcement consiste en général à maximiser globalement la récompense sur un parcours éventuellement infini ou sur un ensemble de parcours finis.

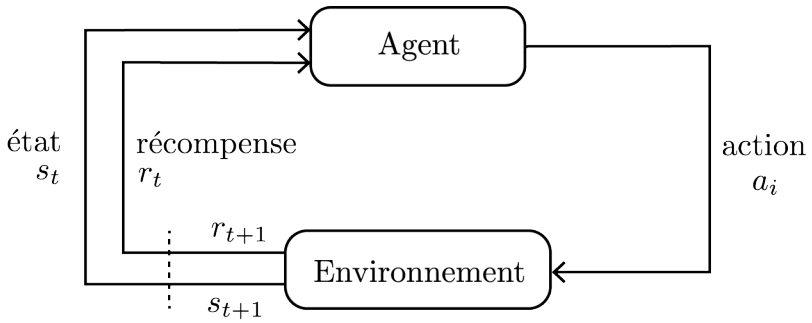

Illustration 1: Apprentissage par renforcement : notions d'environnement, d'agent, d'action, d'état et de récompense.

Afin d'illustrer ces, nous allons considérer le jeu « FrozenLake » (voir illustration 2). Il s'agit d'un jeu implémenté par l'entreprise d'intelligence artificielle OpenAI (https://openai.com/) dans leur framework « Gym » (« Gym is a toolkit for developing and comparing reinforcement learning algorithms » - http://gym.openai.com/). Nous en avons extrait le jeu, pour éviter d'installer ce package Python, il s'agit du fichier « FrozenLake.py » fourni que nous allons utiliser (**sans le modifier!!!**).

Principe du « FrozenLake » : L'agent contrôle le mouvement d'un personnage sur une grille (actions {'Left','Down','Right','Up'} codées par {0, 1, 2, 3}) Certaines cases de la grille sont praticables (cases « S-Start » de départ et « F-Frozen »), et d'autres font tomber l'agent dans l'eau (cases « H-Hole »). De plus, la direction du mouvement de l'agent peut-être incertaine (nous ignorons cet aspect dans un premier temps) et ne dépend que partiellement de la direction choisie. L'agent est récompensé lorsqu'il trouve un chemin praticable jusqu'à la case finale (case « G-Goal »).

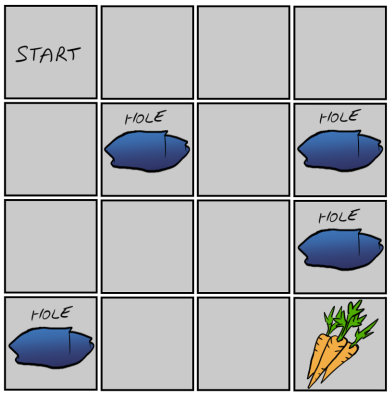

Illustration 2 : Le jeu « FrozenLake ».

Exercice 1 :

1. Récupérer le code fourni (ne pas modifier le fichier « FrozenLake »), exécutez-le et vérifier que vous obtenez la sortie ci-dessous (cf illustration 3).
2. Que se passe-t'il si vous êtes sur une case du haut et que vous chercher à aller en haut ? Quel nouvel état ? Quelle récompense ? Modifier l'exemple fourni pour tester ce scénario et répondre à la question.
3. Coder un scénario gagnant (i.e. ensemble d'action pour aller de S à G sans tomber dans un trou). Quelle est la récompense obtenue en arrivant en G ?
4. Coder un scénario perdant (i.e. ensemble d'action pour aller de S à une case H). Quelle est la récompense obtenue en arrivant sur une case H ?

```
SFFF
FHFH
FFFH
HFFG
==========
 Action [Right] -> New state 1, Reward: 1.00
 Is finished False
SFFF
FHFH
FFFH
HFFG
==========
 Action [Right] -> New state 2, Reward: 1.00
 Is finished False
SFFF
FHFH
FFFH
HFFG
```

```
import FrozenLake as fl

# ENVIRONNEMENT
env=fl.MyFrozenLakeEnv()
env.reset() #Initialization
env.render() #Display


# ACTIONS POSSIBLES
actions={'Left':0,'Down':1,'Right':2,'Up':3}


# AGENT : FAIT DES ACTIONS
for action in (['Right']+['Right']):
    #Recupere nouvel etat (new_state), recompese (reward),
si fini (done)
    new_state, reward, done, _ = env.step(actions[action])
    print('' ...'')
    env.render()
    if done: break
```

Illustration 3 : Sortie console de l'état du jeu et Code associé

3 Un peu de terminologie

Épisode : Un enchaînement complet d'action (e.g. jouer une partie d'un jeu) est associé à la notion d'épisode. Un épisode est l'ensemble fini des états (s), actions entreprises (a) par l'agent et de récompenses (r) reçues:

$$s_0, a_0, r_1, s_1, a_1, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Par exemple, on peut considérer le diagramme de transition qui suit :

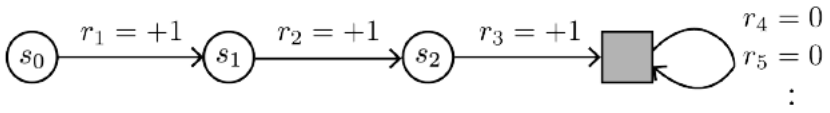

Illustration 2: Exemple de diagramme de transition : on pourrait ajouter les actions a0, a1… sous les flèches

Par exemple, on peut considérer le diagramme de transition qui suit : Dans cet exemple, le carré représente l'état final correspondant à la fin de l'épisode. En parant de s_0, on obtient la séquence de récompense +1; +1; +1; 0; 0; 0; : : :.

Exercice 2 : En récupérant, à chaque étape, l'état courant (**env.s**), l'action et la récompense, afficher, en Python, le diagramme de transition de votre scénario gagnant sous la forme « (s=0,a=2)→r=2→(s=1,a=3)→ ... »

Récompense cumulative : La récompense cumulative est la somme de la récompense accumulée tout au long d'un épisode, qui peut être avec horizon h fini ou infini, avec ou sans rabais (gain cumulé avec intérêt), en supposant que l'on part de l'état s_0 :

- Gain cumulé avec horizon fini h :

$$R = r_1 + r_2 + \dots r_h = \sum_{t=1}^{h} r_t$$

- Gain cumulé avec intérêt et horizon infini :

$$R = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots = \sum_{t=1}^{\infty} \gamma^{(t-1)} r_t$$

où gamma est le rabais compris entre $0 \leq \gamma \leq 1$ 0 et 1 :

Le rabais permet de donner plus de poids aux récompenses dans un futur proche.

Exercice 3 : Calcul en Python et pour votre scénario gagnant (pour chacun des états de l'épisode):

- Le gain cumul infini (l'affichage attendu est indiqué dans le tableau ci-après)
- Le gain cumulé avec un horizon de 3 (sauf pour les 3 derniers états).
- Le gain cumulé infini avec un rabais de 0.9.

state 0 -> cumul infini = 10
state 1 -> cumul infini = 9
state 2 -> cumul infini = 8
state 6 -> cumul infini = 7
state 10 -> cumul infini = 6
state 14 -> cumul infini = 5
Exemple d'affichage attendu pour le cumul infini, pour chaque état de l'épisode.

4 Politique : cas de la Q-Table (Q-learning)

Politique et politique optimale : Une politique est la stratégie utilisée par l'agent pour choisir une action dans chaque état. Elle est notée par π . La politique optimale (π^*) est la politique théorique qui maximise l'attente de la récompense cumulative. Il s'agit

bien de la récompense cumulée : ainsi, pour un état donné, l'action optimale ne sera peut-être pas celle apportant une récompense immédiate (résultant de cette action seulement), mais plutôt celle qui, enchainée à d'autres actions futures, apportera, au total, la meilleure récompense cumulée.

Q-Table : En discret, le système est décrit par des états discrets et des actions discrètes, comme dans notre cas. Dans ce cas particulier, la politique (optimale) peut être définie au moyen d'une Q-Table. Il s'agit tout simplement d'un tableau qui, pour un état donné (ligne du tableau) et une action donnée (colonne du tableau) donne la récompense cumulée (donc la récompense directe cumulée aux récompenses à venir si l'on choisit la meilleure action lors de chacun des états futurs). Naturellement, étant donné que l'on cherche à maximiser la récompense cumulative, on choisira, dans un état *s* donné, l'action *a* optimale, fonction de s ($\pi^*(s)$) maximisant $Q^*(s,a)$:

$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

Exercice 4 : on fournit 2 Q-Tables, que l'on peut charger avec numpy : [**Q=np.load("QTable1.npy")**].

- Ecrivez un script Python vous permettant de « jouer » au FrozenLake en utilisant chaque Q-Table, et en commençant à l'état 0 : en fonction de l'état, vous choisissez l'action optimale selon la Q-Table, et appliquer cette action… ainsi de suite, en répétant cela jusqu'à atteindre l'état 15 (case terminale).
- Laquelle des deux Q-Table est optimale (dans le sens où elle vous emmène à la case finale) ?

Remarque : numpy fournit l'opérateur np.argmax(). Par exemple np.argmax(tab[2,:]) retourne l'indice de la colonne contenant le maximum de la ligne d'indice 2 du tableau [**tab**].

5 Apprentissage par renforcement : cas du Q-learning

L'apprentissage par renforcement consiste à apprendre la politique optimale, donc la fonction Q^* . Cette fonction peut-être déterminée en actualisant itérativement la valeur de Q (Q-Value Iteration), basé sur la relation suivante :

$$Q_{k+1}(s,a) = (1-lr)Q_k(s,a) + lr * [r + y * \max_a(Q_k(s_{t+1},a))]$$

Où γ correspond au poids donné aux récompenses associées aux actions futures (par rapport à la récompense immédiate). Le terme *lr* correspond au "learning rate" associé au pas de mise à jour des valeurs au cours du processus itératif d'apprentissage (typiquement en 0 et 1).

Exercice 5 : Un caneva est fourni pour la procédure d'apprentissage (Ex5Caneva.py). Dans ce script, aucune politique n'est apprise car la Q-Table [**Q**] n'est pas mise à jour et reste dans son état initial. A noter que les récompenses sont toutes nulles ou négatives sauf la case arrivée. Dans le cas contraire les récompenses seront croissante pour la plupart des mouvements sans avoir besoin d'atteindre l'objectif : la plupart des scénarios seront donc gagnant, ne répondant pas à l'objectif attendu (i.e. de pousser à atteindre la case arrivée).

- Ajouter en fin de script vos lignes de codes écrites pour l'exercice 4 et consistant à utiliser la Q-Table pour jouer une partie : vous constaterez que la partie ne termine probablement pas.
- Intégrer la formule précédente (ligne **« TODO: Update Q-Table with new knowledge » - voir tableau ci-dessous**) et vérifier ensuite que la politique optimale est apprise et que la partie s'exécute correctement. Vous pourrez prendre γ =0.95 et lr=0.8.

```
#Environnement
env=fl.MyFrozenLakeEnv(rewards={'S':0,'F':0,'H':-1,'G':5,'NoChange':0})
#Initialization de la Q-Table
Q = np.zeros([env.observation_space.n,env.action_space.n])
#Learning
for i in range(2000): #Pour chaque partie
    s = env.reset() #Reset environment and get first new observation
```

```
    d,j = False,0 #d: True si partie terminated (hole: perdu ou goal: won)
    #The Q-Table learning algorithm
    while j < 99:
        j+=1
        #Select action
        a = env.action_space.sample()
        #Get new state and reward from environment
        s1,r,d,_ = env.step(a)
        #TODO: Update Q-Table with new knowledge
        ??????????
        #Update current state
        s = s1
        if d == True: #Si on termine -> nouvel épisode
            break
```

Script Ex5Caneva.py

Fonction de transition: Souvent, on peut se retrouver avec une probabilité conditionnelle (state-transition probability ou fonction de transition), i.e. probabilité d'arriver dans l'état s_{t+1} sachant que l'état courant est s_t et que l'action engagée est a_t : $P(s_{t+1}|s_t,a_t)$. Dans notre cas, cela peut matérialiser des cases « glissantes » : par exemple on cherche à aller en bas avec le risque de glisser et tomber à gauche ou à droite (avec une certaine probabilité). Dans ce cas, la formulation de la mise à jour de Q diffère, conduisant à l'équation de Bellman

$$Q_{k+1}(s_t,a_t) \leftarrow \sum_{s_{t+1}} P(s_{t+1}|s_t,a_t)\left[r(s_{t+1}|s_t,a_t) + \gamma * \max_a(Q_k(s_{t+1},a))\right]$$

Exercice optionnel : Essayez de jouer plusieurs fois votre scénario appris avec l'option « is_slippery=**True » :** env=fl.MyFrozenLakeEnv(is_slippery=True)…. Observez le comportement aléatoire...

Pour aller plus loin, sachez que nous venons seulement de voir le cas discret avec le Q-learning. Cependant, il existe, même si ces concepts demeurent, d'autres notions et d'autres algorithmes. Par exemple le cas de système continue (i.e. potentiellement nombre « infini » d'états ou d'actions), traités différemment. Dans ce type de situation, la politique est définie non pas par une table mais par une fonction Q(s,a), que l'on peut aborder en utilisant l'apprentissage profond avec des réseaux de neurones (voir Figure 3 ci-après).

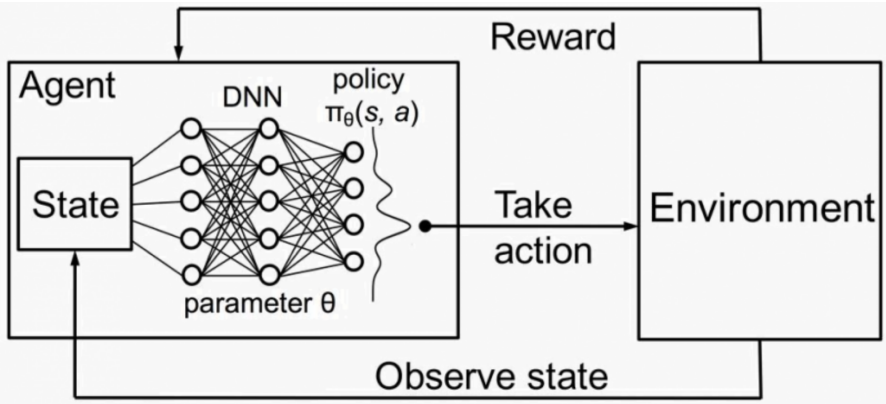

Illustration 3: Apprentissage profond par renforcement (source : https://quantdare.com/deep-reinforcement-trading/)