TME 9 - 10: RL

Nous utiliserons dans ce TME la plateforme en python `gym` (de *open-ai*) ainsi que les paquets `ple` et `gym_ple`. Installez ces 3 paquets (n'oubliez pas de configurer le proxy !) :

```
export https_proxy=proxy:3128
export http_proxy=proxy:3128
pip3 install --user gym
git clone https://github.com/ntasfi/PyGame-Learning-Environment.git
pip3 install  PyGame-Learning-Environment/ --user
git clone https://github.com/lusob/gym-ple.git
pip3 install gym-ple --user
```

Téléchargez également le fichier code source sur le site de l'UE, une fonction `test_gym()` permet de tester que votre installation fonctionne.

Exploration vs Exploitation

Le problème de l'exploration vs exploitation peut se résumer au cas simple suivant (appelé problème du bandit-manchot) : un ensemble $\mathcal{A}$ de n actions a_i sont possibles, chacune procure une récompense moyenne μ_i^* inconnue, comment identifier le plus rapidement l'action la plus profitable ? Vaut-il mieux faire confiance à l'estimation courante et choisir l'action avec la meilleure récompense (option greedy), ou alors chercher une autre action sous-explorée ?
On utilisera les notations suivantes : μ_i^t la moyenne des récompenses reçue pour l'action i au bout de t pas, T_i^t le nombre de fois où l'action a_i a été sélectionnée au temps t (on identifiera l'action et l'indice de l'action).
Les algorithmes que l'on se propose d'étudier sont les suivants :

- uniforme : l'action est choisie uniformément au hasard
- Greedy (glouton) : $argmax_i\ \mu_i^t$
- ϵ-greedy : avec proba ϵ, uniformément sur $\mathcal{A}$; avec proba $1-\epsilon$, $argmax_i\ \mu_i^t$
- Roulette : probabilité de chaque action proportionnelle à la récompense moyenne estimée, $p(a_i) = \frac{\mu_i^t}{\sum_i \mu_i^t}$
- UCB : $argmax\ \mu_i^t + \sqrt{\frac{2\ logt}{T_i^t}}$

Il est courant d'initialiser les algorithmes en jouant d'abord au hasard les actions qui n'ont jamais été jouées jusqu'à ce que toutes les actions aient été testées au moins une fois.

Pour simuler un problème d'exploration/exploitation, une classe `Eve` vous est donnée qui stocke au fur et à mesure les récompenses (dans le tableau `rewards`) et le nombre de fois où chaque action est jouée (dans le tableau `times`). Cette classe est paramétrée lors de son initialisation par une fonction `politique`, qui prend en argument le tableau `rewards` et le tableau `times`, et rend l'action choisie. La classe est également munie de deux méthodes : `getAction()` qui retourne l'action à effectuer et `setReward(action,reward)` qui permet d'indiquer la récompense obtenue lors de l'action choisie. Par défaut, la classe rend un choix uniforme.
Codez les fonctions `greedy(rewards,times)`, `roulette(rewards,times)`, `epsgreedy(rewards,times)`, `ucb(rewards,times)` qui correspondent aux politiques précédentes.
Testez vos fonctions sur un problème de bandit-manchot : sélectionnez n valeurs μ_i^* pour initialiser le problème, puis à chaque pas, demandez à la politique l'action i choisie (par l'intermédiaire de la fonction `getAction()` ; tirez aléatoirement une valeur entre 0 et 1, si celle-ci est inférieure à μ_i, la récompense est de 1, sinon de 0 ; mettez à jour la politique à l'aide de la fonction `setReward(action,reward)`.

Tracez en particulier les récompenses cumulées en fonction du temps pour chaque politique, également les courbes du nombre de fois où chaque numéro est joué. Testez sur différentes valeurs des μ_i^*. Quelle(s) caractéristique(s) sur ces valeurs permettent de contrôler la difficulté de la tâche ? Une politique vous semble-t-elle meilleure qu'une autre ?

Programmation dynamique

L'environnement `Taxi-v2` de `gym` est une tâche de RL où un taxi doit récupérer un client et le déposer à un endroit donné sur une grille de 5 par 5 : (cf `https://gym.openai.com/envs/Taxi-v2/`). Ci-dessous quelques fonctions utiles pour les environnements :

- initialiser un environnement : `game = gym.make('Taxi-v2')` puis `game.reset()`
- `game.action_space` permet de connaître le nombre d'actions possibles ;
- `obs,reward,done,info = game.step(action)` permet de jouer l'action passée en argument : `obs` contient le nouvel état, `reward` la récompense ; associée à l'action, `done` un booléen pour savoir si le jeu est fini ;
- `game.render()` permet de visualiser le nouvel état ;
- `game.observation_space` le nombre d'états
- `game.env.last_action` la dernière action effectuée
- `game.env.s` l'état actuel (codé par un entier)
- `game.env.P` : le mdp de la tâche, sous la forme d'un dictionnaire : chaque clé est un état, chaque valeur un dictionnaire ; pour ce 2ème dictionnaire, chaque clé est une action et la valeur associée une liste de tuples `(proba,état,reward,done)`, la probabilité d'atteindre l'état, la récompense associée et si le jeu est fini ou pas. Ainsi, `game.env.P[state][action]` permet de connaître pour un état et une action la liste des états atteignables avec leur probabilité et la récompense associée.

Codez l'algorithme de policy iteration (rappel : évaluation de la politique : opérateur de Bellman $(T^\pi V)(s_t) = r(s_t, \pi(s_t)) + \gamma V(s_{t+1}))$; amélioration de politique $\pi(s_t) = argmax_a\ r(s_t, a) + \gamma V_\pi(s_{t+1})$, s_{t+1} étant l'état suivant s_t selon la politique π en choisissant donc l'action $\pi(s_t)$). et testez sur l'environnement. Observez la politique apprise sur quelques états.

Q-Learning

Lorsque le MDP n'est pas disponible, on ne peut plus appliquer l'algorithme précédent. Le QLearning utilise des episodes de jeux pour apprendre itérativement la q-value selon la mise-à-jour suivante : $Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1}, +\gamma\ max_a\ Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$. Afin d'obtenir des episodes, il est courant d'utiliser une politique ϵ-greedy en suivant la valeur Q_t estimée.
Codez l'algorithme de Q-Learning et appliquez-le à l'environnement `Taxi`.