TP3 : Q-learning tabulaire

Dans ce TP nous allons mettre en œuvre un algorithme d'apprentissage basé sur l'équation de point fixe de Bellman.

Rappel:

On note : $$Q(s_t, a_t) = E(\sum_{t'=t}^{T_\text{max}} \gamma^{t'-t} r_{t'})$$ la fonction de valeur sur les transitions d'état (avec $\gamma \in [0,1]$.

La fonction de valeur de la politique optimale obéit à l'équation de récurrence : $$Q(s_t, a_t) = r_t + \gamma \sum_{s'} p(s'|s, a) \max_{a'} Q(s', a')$$

Dans le cadre de l'apprentissage par renforcement, les probabilités de transition $p(s'|s, a)$ sont inconnues. On utilise donc une approximation. le principe général du Q-learning est de calculer la valeur optimale à l'aide d'une politique non optimale. Soit $\tilde{\pi}$ cette politique non optimale et s_{t+1} l'état observé après avoir choisi l'action a_t. La valeur de Q optimale est approchée par : $$\tilde{Q}(s_t, a_t) = r_t + \gamma \max_{a'} \tilde{Q}(s_{t+1}, a')$$

Ainsi la valeur de la transition (s_t, a_t) est mise à jour en fonction de valeur calculées précédemment. Les états et les actions étant supposés discrets, les valeurs sont stockées dans une table.

Si les mêmes transitions sont visitées plusieurs fois, il est possible de comparer la valeur précédemment stockée et la nouvelle valeur. La différence entre les deux est appelée *erreur de prédiction*: $$e_t = r_t + \gamma \max_{a'} \tilde{Q}(s_{t+1}, a') - \tilde{Q}(s_t, a_t) $$

La mise à jour du Q-learning repose sur un paramètre d'apprentissage $\alpha \in]0, 1]$ qui fixe la "force" de la mise à jour. $$\tilde{Q}(s_t, a_t) \leftarrow \tilde{Q}(s_t, a_t) + \alpha e_t$$

Algorithme

Dans le cadre du Q-learning, il n'est pas nécessaire de conserver l'historique des récompenses. La mise jour de la valeur est effectuée à chaque nouvelle observation:

```
Q <-- zeros(N_obs, N_act)
s <-- initialiser_environnement()
répéter:
    a <-- choix_selon_politique(s)
    s_prim, r <-- environnement_step(a)
    max_Q_prim <-- max (Q[s_prim,:])
    err <-- r + GAMMA * max_Q_prim - Q[s, a]
    Q[s, a] <-- Q[s, a] + ALPHA * err
    s <-- s_prim
```

Pour les environnements épisodiques, on utilisera une double boucle :

```
Q <-- zeros(N_obs, N_act)
répéter:
```

```
    s <-- initialiser_environnement()
    répéter :
        a <-- choix_selon_politique(s)
        s_prim, r, fini <-- environnement_step(a)
        max_Q_prim <-- max Q[s_prim,:]
        si fini:
            err <-- r - Q[s, a]
        sinon:
            err <-- r + GAMMA * max_Q_prim - Q[s, a]
        Q[s, a] <-- Q[s, a] + ALPHA * err
        s <-- s_prim
        si fini :
            sortir de la boucle
```

Classe agent

La classe agent doit être la plus générique possible pour pouvoir être utilisée dans différents environnements.

On distingue ici deux cas de figure :

- environnement continu : lorsque l'environnement est continu, on doit définir une *discrétisation* de l'environnement. On prendra ici une discrétisation binaire : le nombre d'états discrets est alors 2^N où N est le nombre de dimensions.
- environnement discret : le nombre d'états discrets est donné.

```python
import numpy as np

class Agent:
    def __init__(self, env, ALPHA = 0.1, GAMMA = 0.9):
        self.env = env
        if type(env.observation_space) is gym.spaces.discrete.Discrete:
            self.N_obs = env.observation_space.n
        else:
            dim = np.prod(env.observation_space.shape)
            self.N_obs = 2**dim
        self.N_act = env.action_space.n
        self.Q = np.zeros((self.N_obs, self.N_act))
        self.ALPHA = ALPHA
        self.GAMMA = GAMMA
```

ALPHA et GAMMA sont les 'hyperparamètres' de l'algorithme du Q-learning. Ils doivent être choisis soigneusement pour assurer la bonne convergence de l'algorithme.

Les valeurs par défaut sont ici `ALPHA = 0.1` et `GAMMA = 0.9`

Discrétisation

Pour les premiers tests, nous reprenons l'environnement 'CartPole-v0' du TP2 (pendule inversé). Le pendule inversé étant un environnement à états continus, il faut définir une discrétisation :

```
def discretise(self, obs):
    if type(self.env.env) is gym.envs.classic_control.cartpole.CartPoleEnv:
        return       int(obs[3] >= 0)
               + 2 * int(obs[2] >= 0)
               + 4 * int(obs[1] >= 0)
               + 8 * int(obs[0] >= 0)
    else:
        return obs
```

Choix des politiques

L'algorithme du Q-learning repose sur l'utilisation d'une politique imparfaite (non-optimale) pour trouver les valeurs de Q optimales.

La vitesse et la qualité de la convergence dépendra fortement de la politique choisie. Nous comparerons dans ce TP différentes politiques possibles.

On considérera ici :

- la politique gloutonne ("greedy")
- une politique aléatoire uniforme
- la politique ε-greedy
- la politique softmax

Max et argmax

Écrire une méthode `max_Q` qui retourne la valeur maximale de Q[s,:] (on pourra utiliser la méthode `np.max`).

```
  def max_Q(self, s):
      ...
```

Écrire une méthode `greedy_pol` qui retourne l'action qui maximise Q[s,:] (on pourra utiliser la méthode `np.argmax`).

```
  def greedy_pol(self, s):
      ...
```

Il s'agit ici de la politique "gloutonne" ("*greedy*") d'où le nom.

Politique aléatoire

Écrire une méthode `random_pol` qui retourne une action choisie au hasard dans l'espace des actions possibles. (on pourra utiliser `np.random.randint(self.N_act)`)

```
def random_pol(self, s):
    ...
```

Politique epsilon-greedy

Écrire une méthode `eps_greedy_pol` qui retourne une action choisie au hasard selon la distribution epsilon greedy :

- si $a = \underset{a'}{\text{ argmax }} Q(s,a')$:
  - $\pi(a|s) = 1 - \varepsilon + \frac{\varepsilon}{N_a}$
- sinon:
  - $\pi(a|s) = \frac{\varepsilon}{N_a}$

Ecrire la méthode `eps_greedy_pol` qui choisit une action au hasard selon la distribution définie plus haut.

```
def eps_greedy_pol(self, s, EPSILON = 0.1):
    ...
```

Si `act_probs` est un vecteur de probabilités, le tirage s'effectue avec :

```
act = np.random.choice(len(act_probs), p=act_probs)
```

Politique softmax

La distribution du softmax est calculée selon l'équation de Gibbs: $$\pi(a|s) = \frac{\exp(Q(s,a))}{\sum_{a'}\exp(Q(s,a'))}$$

Pour éviter les problèmes numériques, On utilisera l'algorithme suivant :

```
somme = 0
pour a de 0 à N_act:
    act_probs[a] = exp(Q[s,a] - max_Q(s))
    somme <-- somme + act_probs[a]
act_probs <-- act_probs / somme
```

Ecrire la méthode `softmax_pol` qui choisit une action au hasard selon la distribution du softmax.

```
def softmax_pol(self, s):
    ...
```

Outils d'analyse

Voilà, nous disposons à présent de tous les éléments nécessaires pour écrire l'algorithme du Q-learning!

Le but est de trouver la politique d'exploration et le jeu de paramètres permettant de converger le plus rapidement possible vers la politique optimale. On devra estimer l'effet des différents paramètres :

- ALPHA : plus il est élevé, plus la mise à jour est rapide. Néanmoins, pour des valeurs trop élevées, l'algorithme risque de devenir instable et ne pas converger.
- GAMMA : il représente l'horizon du nombre d'états futurs pris en compte pour le calcul de la valeur. Avec GAMMA = 0.9, le nombre d'états futurs pris en compte est de l'ordre de 10.
- EPSILON : il s'agit du paramètre d'"exploration" de la politique epsilon-greedy. Plus EPSILON est grand, plus la politique se rapproche d'une politique aléatoire, ce qui permet de tester de nombreuses *séquences d'actions*. Plus EPSILON est petit, plus la politique ressemblera à la politique gloutonne qui sert de référence à l'algorithme.

Pour comparer différentes politiques et différents paramètres, il est utile de disposer d'un outil de visualisation. L'apprentissage par récurrence est en effet un processus qui converge lentement du fait de la nature approximative de l'algorithme de mise à jour.

On utilisera ici l'outil TensorboardX qui permet de visualiser certaines variables au cours d'un processus d'apprentissage.

On regardera ici :

- le nombre total d'itérations à chaque fin d'épisode
- la valeur de l'état 0

A chaque exécution de l'algorithme, les variables sont sauvegardées dans un dossier spécifique (qui doit être précisé):

Exemple de programme principal:

```
import gym
from agent import Agent
from tensorboardX import SummaryWriter

ENV_NAME = 'CartPole-v1'
env = gym.make(ENV_NAME)

agent = Agent(env, ALPHA = 0.01, GAMMA = 0.9, CHOICE = 'eps-greedy')
writer = SummaryWriter(log_dir='runs/cartpole/epsgreedy-g_9-a_01')
```

```
N = 1000

for num_episode in range(N):
    agent.run_episode_Q_learning(env, render = False)
    print(num_episode)
    if num_episode % 10 == 0:
        agent.run_episode_glouton(env, render = True)
        print("Test %d done in %d steps" % (num_episode,
env._elapsed_steps))
        writer.add_scalar("#steps", env._elapsed_steps, num_episode)
        writer.add_scalar("V(0)", agent.max_Q(0), num_episode)
```

- Ecrivez la méthode `run_episode_Q_learning` qui exécute un épisode (selon l'algorithme épisodique donné plus haut).
- Ecrivez la méthode `run_episode_glouton` qui exécute un épisode avec la politique gloutonne (sans modifier Q)
- l'argument `render` sert à activer la visualisation graphique dans ces 2 fonctions
- Pensez à ajouter un attribut `CHOICE` dans le constructeur qui détermine le choix de la politique

pour la visualisation

- Pensez à définir un dossier différent pour chaque politique et chaque jeu de paramètres testés
- Pour visualiser les courbes d'apprentissage, ouvrez un terminal à la racine de votre projet et lancez la commande :

```
$ tensorboard --logdir runs
```

- Ouvrez un navigateur. Le tableau de bord est visible à l'adresse : http://localhost:6006.