TP4 : Q-learning sur les espaces d'états continus

L'équation de point fixe de Bellman est ici mise en oeuvre au sein d'un environnement *continu*.

Les espaces d'état discrets offrent un cadre propice au développement d'algorithmes d'approximation de fonctions de valeur grâce à un stockage tabulaire des valeurs et un échantillonnage sur un ensemble discret d'actions.

Néanmoins, de nombreux problèmes de contrôle ont lieu dans des environnements continus. Nous considérons ici le cas où les observations s appartiennent à un espace d'états continu $\mathcal{S}$.

Pour approcher la valeur de transition $Q(s,a)$, il est impossible d'énumérer tous les états possibles dans une table. On utilisera donc un approximateur de fonction paramétrique de type réseau de neurones pour apprendre la fonction de valeur.

Environnement

L'environnement que nous considérons dans ce TP se nomme 'MountainCar' (voir illustration)

original.mp4

.

A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

```
ENV_NAME = 'MountainCar-v0'
env = gym.make(ENV_NAME)
```

Dans cet environnement, l'observation est simplement constituée de deux valeurs continues : la position et la vitesse du véhicule.

Il existe trois actions possibles : pousser à gauche, ne rien faire ou pousser à droite.

La difficulté de l'environnement MountainCar vient du fait que la force appliquée est limitée et que le véhicule doit apprendre à prendre son élan sur le côté opposé pour gravir la montagne (un peu comme une balançoire).

La récompense est ici de -1 à chaque pas de temps (le signe de la récompense est inversé par rapport au pendule du TP3). Ici, on cherche à obtenir les épisodes les plus courts possibles (et non les épisodes les plus longs).

Le simulateur stoppe la simulation après 200 pas de temps. Il y a donc deux conditions d'arrêt :

- le véhicule a atteint l'objectif pour $t < 200$
- le véhicule n'a pas atteint l'objectif et $t = 200$

Pour calculer l'erreur de prédiction, doit prendre en compte ce cas de figure, on modifie le calcul de

l'erreur comme suit :

```
        if done and self.env._elapsed_steps < 200:
            err = reward - self.Q[s, a]
        else:
            err = reward + self.GAMMA * self.max_Q(s) - self.Q[s, a]
```

Dans un premier temps, essayez de résoudre cet environnement à l'aide du Q-learning tabulaire vu au TP3. Cela implique de modifier un peu la méthode de discrétisation (voir TP3) pour prendre en compte ce nouvel environnement.

Ici la position de départ étant à -0.5, le seuil de la discrétisation de la position est fixé à -0.5:

```
def discretise(self, obs):
    if type(self.env.env) is ...:
        ...
    elif type(self.env.env) is
gym.envs.classic_control.mountain_car.MountainCarEnv:
        return       int(obs[0] >= -0.5)
               + 2 * int(obs[1] >= 0)
    else:
        ...
```

créez dans le projet du TP3 un nouveau programme principal `main_mountain.py`. Initialisez l'environnement `MountainCar-v0` et exécutez l'apprentissage sur plusieurs valeurs ce ALPHA et GAMMA pour essayer d'optimiser le comportement de l'algorithme.

Mise en oeuvre dans un environnement continu

La prise en compte d'états continus nécessite de définir la fonction de valeur Q comme une fonction paramétrique dont les paramètres sont mis à jour par descente de gradient sur l'erreur de prédiction (voir le cours sur les réseaux de neurones).

Pour conserver l'environnement virtuel mis en place dans le TP précédent, définissez une nouvelle classe `Agent_nn` au sein du même projet.

Nous utiliserons ici la librairie pytorch pour implémenter le réseau de neurones.

```
import torch
import torch.nn as nn
```

Pensez à installer la librairie torch dans l'environnement virtuel de votre projet

Voici la forme que prend la nouvelle classe Agent

```
class Agent_nn:
    def __init__(self, env, GAMMA = 0.9, ALPHA = 0.001, CHOICE =
'epsgreedy'):
        self.env = env
        self.N_obs = np.prod(env.observation_space.shape)
```

```
        self.N_act = env.action_space.n
        self.GAMMA = GAMMA
        self.ALPHA = ALPHA
        self.CHOICE = CHOICE
        self.net = nn.Sequential(
            nn.Linear(self.N_obs + self.N_act, N_HIDDEN),
            nn.ReLU(),
            nn.Linear(N_HIDDEN, 1)
        )
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr =
self.ALPHA)
```

L'agent a maintenant pour attributs supplémentaires

- un réseau de neurones `self.net` à deux couches, avec une fonction d'activation non-linéaire ReLu sur la couche intermédiaire, et une sortie linéaire.
- un optimiseur permettant de calculer les valeurs de gradient par rétropropagation sur toutes les couches
  - ici la méthode Adam est sélectionnée. On pourra également tester la méthode SGD.
  - `lr` est le *learning rate*.

On pourra fixer la valeur de N_HIDDEN à 50.
La définition d'un réseau de neurones en pytorch repose sur une structure de données spécifique (un tenseur torch)

- qui contient eut être stocké sur CPU ou sur GPU
- qui contient, en plus des paramètres, des valeurs de gradient.

Toutes les données manipulées par le réseau de neurone doivent être au format torch.

Pour passer d'un format numpy à un format torch :

```
x_tf = torch.FloatTensor([x])
```

Pour passer d'un format torch à un format numpy :

```
x = x_tf.detach().numpy()
```

On trouve sur le Web de nombreux tutoriels

remarque: la plupart des fonctions numpy sont implémentées dans l'environnement torch (torch.sin, torch.pow, torch.prod, torch.zeros, etc...)

Définir la fonction Q

La fonction de valeur Q devient une méthode de classe :

```
def Q(self, obs, act, tf = False):
    input = np.concatenate((obs, self.one_hot(act)))
    input_tf = torch.FloatTensor([input])
    output_tf = self.net(input_tf)
```

```
    if tf:
        return output_tf
    else:
        return output_tf.data.numpy()[0]
```

On utilise pour coder l'action un encodage unaire ("*one hot encoding:*")

A faire

Ecrivez la méthode :

```
def one_hot(self, act):
    ...
```

qui retourne un vecteur de taille N_a contenant des zéros partout sauf dans la case d'indice `act`.

A faire : il faut maintenant reprendre la plupart des fonctions et des politiques définies dans le TP3 :

- en éliminant la discrétisation
- en remplaçant Q[obs, act] par Q(obs, act)

Fonction de perte

Il reste bien sûr le plus important qui est la définition de la mise à jour du réseau de neurones en fonction de l'erreur de prédiction.

La fonction de perte est définie ici comme le carré de l'erreur de prédiction. La dérivée de la fonction de perte est l'erreur de prédiction qui est rétropropagée dans l'ensemble du réseau pour mettre à jour les poids par descente de gradient.

```
    def Q_loss(self, act, output_tf, obs_prim, reward, done): # Q TD_error
        Q_prim = self.max_Q(obs_prim)
        if done and self.env._elapsed_steps < 200:
            Q_target = reward
        else:
            Q_target = reward + self.GAMMA * Q_prim
        Q_target_tf = torch.FloatTensor([Q_target])
        return torch.sum(torch.pow(Q_target_tf - output_tf, 2), 1)
```

Algorithme

La mise à jour de la fonction de valeur se fait à chaque pas de temps comme dans l'algorithme tabulaire:

```
    def run_episode_Q_learning(self, env, render = False):
        obs = env.reset()
        obs[1] *= 100
        while True:
```

```
            if render:
                env.render()
            if self.CHOICE == 'softmax':
                act = self.choix_action_softmax(obs)
            else:
                act = self.choix_action_epsgreedy(obs)
            obs_prim, reward, done, _ = env.step(act)
            obs_prim[1] *= 100
            output_tf =  self.Q(obs, act, tf = True)
            loss = self.Q_loss(act, output_tf, obs_prim, reward, done)
            loss.backward()
            self.optimizer.step()
            self.optimizer.zero_grad()
            obs = obs_prim
            if done:
                break
```

NB : l'instruction `obs[1] *= 100` sert à normaliser les vitesses pour faciliter l'apprentissage car les valeurs de vitesse ne sont pas du même ordre de grandeur que les valeurs de position.

Pour rendre le code plus générique, il est conseillé de définir une fonction de normalisation des entrées.

```
def normalise(obs)
    if type(self.env.env) is gym.envs.classic_control.cartpole.CartPoleEnv:
        obs[0] *= 10
        obs[2] *= 10
    elif type(self.env.env) is
gym.envs.classic_control.mountain_car.MountainCarEnv:
        obs[1] *= 100
    return obs
```

A faire Pour faire fonctionner l'algorithme sur le probème MountainCar, il est nécessaire

- de paramétrer finement les valeurs d'ALPHA et de GAMMA.
- de faire tourner l'apprentissage sur un nombre d'épisodes très élevé

On pourra commencer par tester l'`Agent_nn` sur l'environnement Cartpole qui converge plus facilement vers la solution.

Afin de faciliter la résolution du problème, reprendre l'outil de visualisation TensorboardX pour visualiser l'évolution de certaines variables au cours de l'apprentissage.

On regardera ici :

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

```
writer = SummaryWriter(log_dir='runs/%d/%d-g-%.5f-a-%.5f' % (ENV_NAME,
CHOICE, GAMMA, ALPHA))
```