1 Environnement virtuels

La solution la plus simple est d'utiliser un environnement virtuel. Plusieurs choix s'offrent à vous : virtualenv + pip, pipenv ou anaconda. Un environnement virtuel permet d'isoler les dépendances python du projet des dépendances python installées dans le système ou dans les autres projets. Cela évite de nombreux conflits de version entre vos différents projets.
Si vous souhaitez utiliser virtualenv :

```
python3 -m venv drl
source drl/bin/activate
pip3 install gym torch
```

1.1 Introduction à Gym

En apprentissage par renforcement, il y a deux concepts fondamentaux ; l'agent et l'environnement. L'agent est l'entité apprenante qui observe l'environnement et agit sur celui-ci selon les actions disponibles. Son objectif est de maximiser la récompense cumulée qu'il reçoit de l'environnement avec lequel il interagit. Chaque environnement a :

- Un espace d'action.
- Un espace d'état.
- Une fonction de récompense.

Gym (https://github.com/openai/gym) propose une interface unifiée entre un agent et un environnement. Ainsi, il est possible de construire indépendamment un agent de l'environnement et inversement ; il suffit alors de satisfaire l'interface pour qu'un nouvel environnement soit compatible avec presque tous les agents précédemment codés pour d'autres environnements. Lorsque certains pré-traitements sont nécessaires sur les actions, observations ou récompenses, il est possible d'encapsuler l'environnement dans un wrapper, celui-ci se chargera du pré-traitement. De plus gym a un méchanisme de registre qui permet de construire un environnement simplement avec son nom.

Dans notre cas nous allons implémenter un agent que l'on pourra ensuite tester sur plusieurs environnements différents.

Il existe plusieurs fonctions clés pour interagir avec un environnement. Un exemple très simple d'agent agissant de manière aléatoire dans un environnement peut être :

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
```

```
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

Une liste de tous les environnements est disponible sur https://github.com/openai/gym/wiki/Table-of-environments

2 Deep Q-network sur CartPole

2.1 Début

L'algorithme que nous allons implémenter est décrit dans [Mnih et al., 2015]. Plus de précisions et d'aides sont disponibles dans l'article. Nous l'appliquerons à la tâche CartPole-v1, qui consiste à garder un bâton levé le plus longtemps possible.

Question 1. *En vous inspirant du lien donné ci-dessus, faites interagir un premier agent de manière aléatoire avec l'environnement CartPole-v1.*

Question 2. *Pour évaluer notre agent, nous aurons besoin de suivre l'évolution de la récompense obtenue à chaque épisode. Proposez une manière de suivre l'évolution de la somme de récompense par épisodes en fonction du nombre d'interaction de l'agent avec l'environnement. Une manière de faire serait d'afficher des logs et/ou d'afficher la courbe à la fin de l'apprentissage via matplotlib.*

Rappel : Un épisode débute lors du reset et se termine lorsqu'il reçoit un signal de fin de la part de l'environnement.

2.2 Experience replay

Nous allons maintenant pouvoir commencer l'implémentation de notre algorithme. Le DQN utilise l'experience replay, i.e., il stocke dans en mémoire toutes les interactions qu'il reçoit, où interaction=(état,action,étatsuivant,récompense,finépisode). Le buffer a une taille maximale (100 000 par exemple) ; lorsqu'elle est dépassée, les nouvelles expériences remplacent les plus anciennes. L'agent va apprendre des expériences stockée dans son buffer, il choisira aléatoirement un minibatch d'expériences dans son buffer.

Question 3. *Implémentez le buffer et stockez les données reçues. Faites attention à bien gérer le dépassement de la taille maximale. Le buffer doit être indépendant de l'environnement, et donc de l'espace d'action et d'état.*

Question 4. *Implémentez une fonction de sampling permettant de récupérer un minibatch de données aléatoires dans le buffer.*

2.3 Deep Q-learning

Nous allons utiliser des réseaux de neurones pour approximer l'espérance de récompense cumulée $Q(s, a)$. Ensuite nous appliquerons l'équation de Bellman.

Question 5. *Construisez un réseau de neurone qui, prenant en entrée un état, renvoie la Q-valeur de toutes les actions.*

Le réseau prend un entrée un état, donc la première couche doit être de même taille que l'état. La sortie est d'une Q-valeur par action, donc la couche de sortie doit être de même taille que le nombre d'actions possibles.

Question 6. *Plutôt que de choisir des actions aléatoires, nous pouvons utiliser cette approximation de Q pour choisir les actions. A chaque nouvel état que reçoit votre agent, calculez la Q-valeur des actions. Choisissez également une stratégie d'exploration pour découvrir de nouveaux états.*

Nous proposons deux stratégies classiques d'exploration.

La méthode $\epsilon - greedy$ permet de choisir avec une probabilité ϵ, une action aléatoire, et avec une probabilité $(1 - \epsilon)$ la meilleure action. Cette stratégie a l'avantage d'être simple.

L'exploration Boltzmann permet de choisir une action a_k selon la probabilité :

$$P(s, a_k) = \frac{e^{\frac{Q(s,a_k)}{\tau}}}{\sum_i e^{\frac{Q(s,a_i)}{\tau}}}. \tag{1}$$

où τ un hyper-paramètre désignant la stochasticité de l'exploration, plus il est grand, plus la politique sera aléatoire, plus il est faible, plus la politique tendra vers une politique déterministe maximisant la Q-valeur actuelle. Intuitivement, l'exploration Boltzmann pondère la probabilité d'une action selon sa Q-valeur.

Question 7. *Pour l'instant, notre agent n'apprend pas. Après chaque interaction, choisissez aléatoirement dans votre buffer un minibatch de données, calculez les Q-valeurs, appliquez*

l'équation de Bellman et rétropropagez l'erreur dans le réseau de neurone.

Rappel :

- $J_\theta = \left[Q_\theta(s,a) - (r(s,a) + \gamma \max_{a'} \hat{Q}_\theta(s',a'))\right]^2$ si l'épisode continue.
- $J_\theta = [Q_\theta(s,a) - r(s,a)]^2$ si l'épisode se termine.

A partir de là, il se peut que votre agent apprenne des comportements intéressants mais qu'ils soient très instables. C'est dû au fait que la fonction d'erreur n'est pas stationnaire, modifier $Q_\theta(s,a)$ modifie également $\hat{Q}_\theta(s',a)$; une backpropagation du gradient peut alors augmenter l'erreur.

Question 8. *Nous allons utiliser un target network pour stabiliser l'apprentissage. Clonez votre réseau de neurone et utilisez le duplicat par calculer la Q-valeur ciblée. Mettez à jour de manière graduelle le duplicat vers le réseau de neurone original.*

Il y a deux manières possible de mettre à jour le target network :

- Toutes les N étapes d'apprentissage (10000 par exemple), recopier entièrement le réseau de neurone original dans le duplicat.
- Mettre à jour petit à petit le duplicat à chaque étape d'apprentissage : $\theta' = (1-\alpha)\theta' + \alpha\theta$ où θ' sont les poids du duplicat, θ les poids du réseau original, et α le pas de mise à jour. Souvent $\alpha = 0.005$ ou $\alpha = 0.01$.

Dorénavant, votre DQN est censé fonctionner et votre agent devrait réussir à manipuler le CartPole !

Références

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540) :529.