Utilisation de la librairie OpenAI Gym

Comme nous l'avons vu au TP précédent, l'apprentissage par renforcement repose sur l'interaction entre un agent et un environnement:

- L'environnement correspond à la fois à l'environnement physique et aux dispositifs matériels. Dans le cadre de l'apprentissage par renforcement, on utilise souvent un environnement simulé (qui reproduit le comportement de l'environnement physique)
- L'agent correspond à un opérateur logiciel capable de percevoir les états de l'environnement par ses capteurs et d'agir sur l'environnement grâce à ses actionneurs.

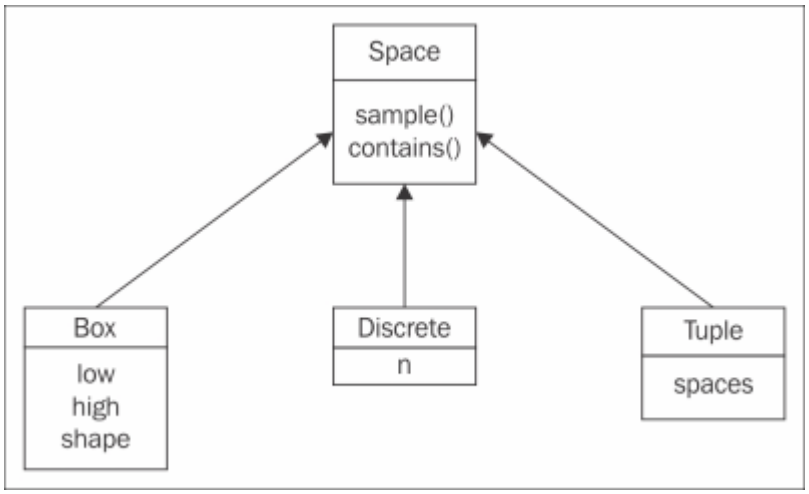
On considère de plus que le temps s'écoule de façon discrète et que l'état de l'environnement au temps t peut être décrit à l'aide d'un vecteur d'état $\boldsymbol{s}_t$.

Dans le cas parfaitement observable, l'état de l'environnement est égal au vecteur d'observation.

La librairie `Gym` a été développée par OpenAI (voir http://www.openai.com). Elle propose une riche collection d'environnements pour les expériences d'AR à l'aide d'une interface unifiée.

Classe Space

La classe `Space` définit un domaine de valeurs (discretes ou continues) que peuvent prendre les actions ou les observations.



Un espace peut être de type discret, continu, ou encore être défini comme une combinaison d'espaces discrets ou continus. Tous les espaces implémentent les méthodes :

- `sample()` : retourne une valeur tirée aléatoirement dans le domaine de valeurs
- `contains(x)` : indique par 'True' ou 'False' si `x` appartient au domaine de valeurs.

- Un espace de type `Discrete` est décrit par le nombre `n` de valeurs possibles qu'il peut prendre;
- Un espace de type `Box` est un espace vectoriel borné :
  - l'attribut `shape` donne la dimension
  - l'attribut `low` donne la borne inférieur sur les différents axes
  - l'attribut `high` donne la borne supérieure sur les différents axes.

Classe Env

La classe centrale est la classe Env. elle dispose de plusieurs méthodes et attributs qui apportent les informations nécessaires à la mise en œuvre de l'apprentissage.

La classe Env possède deux attributs de type Space:

- `action_space` : le domaine des valeurs d'actions autorisées dans cet environnement
- `observation_space` : le domaine des valeurs d'observations possibles sur cet environnement

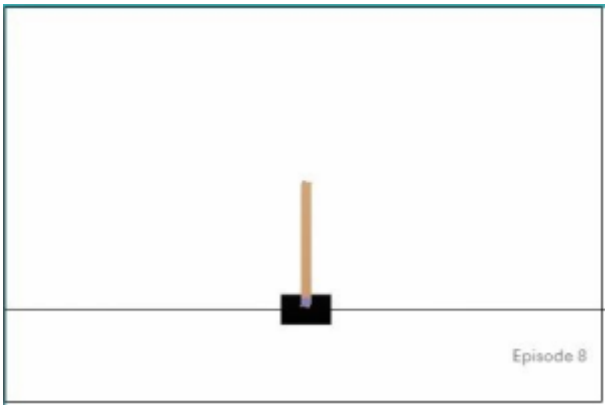
Elle propose les méthodes:

- la méthode `step()` permettant d'exécuter des actions, et qui retourne:
  - l'observation courante
  - la récompense
  - une indication pour savoir si l'épisode est fini
- la méthode `reset()` qui ramène l'environnement à son état initial et fournit la première observation.
- la méthode `render()` qui permet de visualiser l'état de l'environnement à l'aide d'une interface graphique.

remarque : la méthode `step()` est la pièce centrale de l'environnement, correspondant à l'itération de l'état de l'environnement entre deux actions produites par l'agent. Ainsi, pour utiliser l'environnement, il suffit d'écrire une boucle qui alterne les appels à la méthode `step()` de l'environnent et le choix de l'action par l'agent. Bien sûr, si le comportement de l'environnement est entièrement géré par Gym, la définition du comportement de l'agent est à la charge du programmeur.

Le pendule inversé

Dans ce TP, nous nous intéressons au problème du pendule inversé (voir https://gym.openai.com/envs/CartPole-v0).



Une barre rigide est attachée par une charnière à un chariot, qui bouge le long d'un rail sans friction.

Le système est contrôlé en appliquant une force de +1 ou -1 au chariot.

Le pendule démarre en position verticale instable, et le but est de le maintenir ainsi le plus longtemps possible, en appliquant des forces de +1 ou -1 alternativement.

Une récompense de +1 est donnée tant que le pendule est maintenu en position verticale.

L'épisode s'arrête dès que:

- le pendule est à plus de 15 degrés de la verticale
- le chariot est à plus de 2.4 unités du centre

Le vecteur d'observation de cet environnement est constitué de quatre nombres réels donnant la position du chariot, sa vitesse, la coordonnée angulaire du pendule et sa vitesse angulaire.



NB : si on connaît les caractéristiques physiques du chariot, il est possible de trouver une loi de contrôle qui maintient le pendule en position verticale. Le problème d'apprentissage consiste à apprendre à maintenir le pendule en position verticale sans connaître la signification des grandeurs mesurées.

Premières simulations



Pour installer gym dans Pycharm, il faut utiliser un environnement virtuel (qui donne le droit d'installer les librairies non présentes dans l'environnement par défaut).

- Pour configurer un environnement, suivre ce lien.
- une fois l'environnement sélectionné, cliquer sur le petit '+' vert sélectionner la librairie gym dans la liste.
- l'installation peut prendre un peu de temps...

Chaque environnement proposé par Gym possède un nom unique.

Pour voir la liste des environnements proposés, suivre https://gym.openai.com/envs. Le pendule inversé appartient à la catégorie des problèmes de contrôle classiques. Il fait partie des problèmes servant à étalonner algorithmes d'apprentissage par renforcement.

L'environnement du pendule inversé s'appelle 'CartPole-v1'

Pour créer l'environnement :

```
import gym
env = gym.make('CartPole-v1')
```

L'objet env est notre simulateur d'environnement. La commande :

```
obs = env.reset()
```

initialise l'environnement et retourne la première observation, qui est comme nous l'avons vu constituée de quatre nombres réels.

L'état de l'environnement peut être visualisé grâce à la commande

```
env.render()
c = input('Terminer?')
```

remarque : l'ajout de l'invitation 'Terminer' permet de maintenir la fenêtre active. La fenêtre disparaît lorsque la simulation se termine.

Il est possible de modifier la position du pendule à l'aide d'une action. Pour choisir une action au hasard:

```
action = 0
```



NB : L'espace des actions est limité à deux valeurs possibles : 0 ou 1

- l'action 0 pousse le chariot vers la gauche
- l'action 1 pousse le chariot vers la droite

pour appliquer l'action :

```
observation, reward, done, _ = env.step(action)
env.render()
c = input('Terminer?')
```

On peut remarquer sur la fenêtre graphique que le pendule a bougé de manière infinitésimale suite à l'application de l'action.



NB : la fonction step retourne plusieurs valeurs :

- une observation
- un reward (recompense)
- un indicateur done qui vaut True lorsque l'épisode est terminé et False sinon

Pour le faire bouger de manière plus franche, il faut appliquer plusieurs fois l'action. Essayons de l'appliquer 100 fois :

```
obs = env.reset()
for i in range(100):
    env.render()
    action = 0
    obs, reward, done, _ = env.step(action)
c = input("Terminer?")
```

Le chariot disparaît très vite à gauche de l'écran (l'inverse se produit si on applique uniquement l'action 1)

Pour contrôler le pendule, il faut donc alterner les actions 0 et 1. Essayons :

```
obs = env.reset()
```

```
for i in range(1000):
    env.render()
    if i % 2 == 0 :
        action = 0
    else:
        action = 1
    obs, reward, done, _ = env.step(action)
c = input("Terminer?")
```

C'est un peu mieux, mais la barre ne reste pas en position verticale (le système est instable)

Essayons de prendre en compte l'état de l'environnement : un coup à droite si la position angulaire est négative et un coup à gauche si la position angulaire positive:

```
obs = env.reset()
for i in range(1000):
    env.render()
    if obs[2] < 0:
        action = 0
    else:
        action = 1
    obs, reward, done, _ = env.step(action)
c = input("Terminer?")
```

C'est encore un peu mieux mais le pendule finit quand même par se déstabiliser.

Si nous prenons en compte l'indicateur de fin d'épisode `done`, nous voyons que la simulation s'arrête en fait assez rapidement:

```
obs = env.reset()
total_steps = 0
while True:
    env.render()
    if obs[2] < 0:
        action = 0
    else:
        action = 1
    obs, reward, done, _ = env.step(action)
    total_steps += 1
    if done:
        break
print("Episode terminé après %d itérations" % total_steps)
c = input("Terminer?")
```

Il est également possible de contrôler le pendule de manière aléatoire à l'aide le la méthode `sample()` de l'espace d'actions, mais le résultat n'est guère probant:

```
obs = env.reset()
total_steps = 0
while True:
    env.render()
```

```
    action = env.action_space.sample()
    obs, reward, done, _ = env.step(action)
    total_steps += 1
    if done:
        break
print("Episode terminé après %d itérations" % total_steps)
c = input("Terminer?")
```

A faire

- Modifiez le code pour que ce soit l'utilisateur qui choisisse à chaque pas de temps l'action à produire. Quel score obtenez vous?
- Définissez une loi de contrôle qui prenne en compte à la fois la position angulaire `obs[2]`et la vitesse angulaire `obs[3]`.

Discrétiser l'environnement

Les algorithmes d'apprentissage que nous avons vus jusqu'à présent reposent sur de états discrets. On parle d'approche *tabulaire*.

Définir un critère simple : `obs[2] < 0` pour choisir l'action revient à discrétiser l'environnement, en deux états :

- l'état 0 : `obs[2] < 0`
- l'état 1 : `obs[2] >= 0`

On peut généraliser ce principe et définir 16 états différents à partir des 4 observables :

- état 0 : `obs[0] < 0`, `obs[1] < 0`, `obs[2] < 0`, `obs[3] < 0`
- état 1 : `obs[0] >= 0`, `obs[1] < 0`, `obs[2] < 0`, `obs[3] < 0`
- ...
- état 15 : `obs[0] >= 0`, `obs[1] >= 0`, `obs[2] >= 0`, `obs[3] >= 0`

**A faire ** : écrire une fonction qui calcule un état discret à partir des valeurs d'une observation.

Calcul d'une fonction de valeur sur les transitions

L'apprentissage par renforcement consiste ici à procéder par essai/erreur jusqu'à obtenir des durées de simulation les plus longues possibles. Chaque pas de temps apporte une récompense de 1. Les expériences plus longues apportent donc une récompense cumulée plus importante

Rappel : Si s_t est l'état à l'instant t : $$V(s_t) = E(\sum_{t'=t}^T r_{t'})$$ est la fonction de valeur de l'état (avec T l'instant final (fin de l'épisode)). Ainsi les épisodes les plus longs apportent une récompense cumulée plus importante.

La "fonction de valeur" d'un état estime la *récompense cumulée moyenne* obtenue à partir de cet état à partir d'un grand nombre d'observations.

Contrairement au TD1, nous ne possédons pas de modèle de transitions d'état. La fonction de valeur sera ici estimée directement sur les transitions d'état (s_t, a_t) pour permettre de définir une

politique qui maximise *la valeur de la transition*. On note : $$Q(s_t, a_t) = E(\sum_{t'=t}^T r_{t'})$$ la fonction de valeur sur les transitions d'état.

NB : l'état s_t est un état *discret*.

Il est nécessaire pour cela d'ajouter à l'agent une liste les états/actions visités au cours de l'épisode. Cette liste est appelée la `trace`.

Nous reprenons ici le modèle d'agent vu dans le TD1:

```
class Agent:
    def __init__(self):
        self.total_reward = 0.0
        self.trace = []
```

(pas besoin de méthode `step()` ici)

Cette trace doit être mise à jour à chaque itération pour obtenir en fin d'épisode la liste de tous les couples(état, action) parcourus.

On suppose ici que l'agent choisit ses actions au hasard:

```
action = env.action_space.sample()
```

Pour calculer (de façon simplifiée) la fonction de valeur, nous utiliserons l'algorithme suivant :

```
Q = {}
nb_visites = {}
total_rewards_sum  = {}
pour tout couple (s,a) possible:
   Q[(s,a)] = 0
   nb_visites[(s,a)] = 0
   total_rewards_sum[(s,a)] = 0
pour i de 1 à N:
    exécuter un épisode
    pour tout (s,a) dans agent.trace :
        nb_visites[(s,a)] += 1
        total_rewards_sum[(s,a)] += agent.total_reward
pour tout (s,a) dans Q:
    si nb_visites[(s,a)] > 0:
        Q[(s,a)] = total_rewards_sum[(s,a)] / nb_visites[(s,a)]
```

Pour calculer la fonction de valeur, on utilisera N = 1000

Modifier l'init :

```
class Agent:
    def __init__(self):
        self.total_reward = 0.0
        self.trace = []
        self.Q = {}
```

```
        self.nb_visites = {}
        self.total_rewards_sum  = {}
        ...
```



L'initialisation doit également inscrire des valeurs nulles dans les dictionnaires:

```
pour tout couple (s,a) possible:
  Q[(s,a)] = 0
  nb_visites[(s,a)] = 0
  total_rewards_sum[(s,a)] = 0
```

Définir les méthodes suivantes :

- une méthode qui exécute un épisode complet en utilisant une politique aléatoire:

```
def run_episode_random(self, env):
    ...
```



Attention, la `trace` doit contenir des couples (état_discret, action), avec l'état discret calculé à l'aide de la fonction précédente.

- une méthode qui met à jour les dictionnaires `nb_visites` et `total_rewards_sum` en fin d'épisode (lorsque la `trace` et `total_reward` sont connus):

```
def update_dicos(self):
    ...
```



`Attention penser à remettre zéro la trace et total_rewards une fois update_dicos exécuté!`

- une méthode qui estime la valeur de Q après un grand nombre d'épisodes :

```
def update_Q(self):
    ...
```

Implémentez ensuite l'algorithme indiqué à l'aide de ces trois méthodes.

Politique guidée par la valeur

Pour améliorer sa récompense moyenne, l'agent va maintenant prendre en compte la *valeur de la transition d'état* pour choisir l'action, autrement dit, si s_t est l'état courant : $$ a_t = \underset{a \in \mathcal{A}}{\text{argmax }} Q(s_t, a)$$

- Écrire une méthode `choix_action()` dans laquelle l'action est choisie selon la fonction de

valeur (et non plus aléatoirement)

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

- Lancer des simulations avec cette nouvelle méthode :

```
action = agent.choix_action(s)
```

- et calculer la récompense moyenne obtenue sur 100 épisodes et comparer au résultat précédent. Conclusion?

Améliorations

Le calcul exact de la fonction de valeur nécessite de mémoriser l'historique complet des récompenses. Au lieu de conserver la somme des récompenses, on conserve la liste des récompenses obtenues.

```
class Agent:
    def __init__(self):
        self.rewards = []
        self.trace = []
        self.Q = {}
        self.nb_visites = {}
        self.total_rewards_sum  = {}
        ...
```

et la mise à jour se fait en ajoutant les récompenses en fin de liste :

```
self.rewards += [reward]
```

à chaque pas de temps (après chaque appel à `env.step()`).

En fin de simulation, la valeur cumulée est calculée séparément pour chaque (s,a) rencontré:

- pour chaque couple s_t, a_t de la trace :
  - calculer la somme des récompenses **présente et futures**
  - ajouter la valeur dans `total_rewards_sum`
  - incrémenter le nombre de visites

Calculer à nouveau la fonction de valeur et comparer la politique obtenue à la politique précédente.

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/public:rl_tp2

Last update: **2019/01/07 16:07**