Travaux Pratique : Implémentation du Q-learning (sur Cart Pole)

Abdoulaye DIOP
abdoulaye.diop@irit.fr

Robin CUGNY
robin.cugny@irit.fr

Max CHEVALIER
max.chevalier@irit.fr

January 26, 2023

Dans ce TP nous nous intéressons au problème du "Cart Pole" et à l'implémentation de l'algorithme de Q-learning pour le résoudre.

Dans un premier temps, nous nous intéressons à la librairie de simulation, *Gym*, en rappelant les éléments de bases de la librairie. Nous nous familiariserons ensuite avec l'environnement "Cart Pole" que propose Gym avant de passer à l'implémentation de l'algorithme de Q-learning.

1 Introduction à Gym

Gym est une librairie qui permet de simuler des environnements pour le reinforcement learning. La librairie dispose de plusieurs outils permettant de créer des environnements, d'utiliser des environnements pré-existants et d'interagir avec ces derniers.

1.1 Les Spaces

Les spaces sont les objets de base que l'on manipule dans Gym. Ils permettent de représenter l'espace des observations possibles d'un environnement et les actions qu'un agent peut exécuter dans cette environnement. Voici quelques exemples de spaces.

- `Box` : décrit les espaces continus en n dimensions. Chaque dimension a une borne inférieure et une borne supérieure. Par exemple pour générer un espace à 3 dimensions chacune bornée entre 0 et 1:

```
>>> from gym import spaces
>>> import numpy as np
>>> spaces.Box(low=np.array([0, 0, 0]), high=np.array([1, 1, 1]))
Box([0.0, 1.0, (3,), float32)
```

  Les paramètres `low` et `high` doivent être spécifiés sous forme de de tableaux numpy. C'est pourquoi on utilise la fonction `np.array()` qui permet de transformer une liste en tableau numpy (`array`)

- `Discrete` : décrit les espaces discrets de la forme {0, 1, ..., n-1}. Syntaxe :

```
>>> spaces.Discrete(2)
Discrete(2)
```

- `Dict` : dictionnaire d'espaces simples (Box, Discrete)
- `Tuple` : tuple d'espaces simples (Box, Discrete)

Pour mieux comprendre les spaces, Considérons un jeu où le joueur peut se déplacer sur une aire rectangulaire au sol de 5m de longueur et 3m de large.

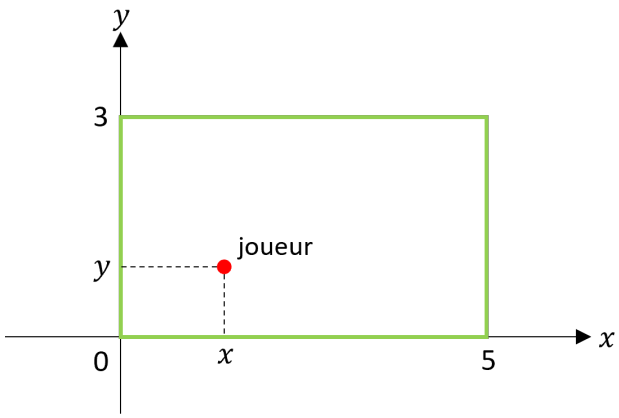

Figure 1: Joueur se déplaçant sur une aire rectangulaire

1. On représente la position du joueur sur cette aire par ses coordonnées (x, y) telles que $x \in [0, 5]$ et $y \in [0, 3]$. Créer un objet de type `Box` représentant l'espace des positions (x, y) que le joueur peut occuper.
2. Observer les attributs `low` et `high` de cet objet en les affichant sur la console.
3. Chaque space dispose d'une méthode `sample` qui permet de générer aléatoirement une observation dans l'espace qu'il défini.

   Utiliser cette méthode pour générer une position aléatoire du joueur sur l'aire de jeu. Observer la position obtenue et vérifier que cette position est bien dans l'aire définie initialement
4. On considère que le joueur peut se déplacer dans 4 directions (haut, bas, gauche et droite). Créer un objet représentant l'espace des actions que peut réaliser le joueur en choisissant le type de space adéquat.
5. Observer le résultat de la méthode `sample` pour ce nouvel objet en l'affichant sur la console.

1.2 Les Environnements

1.2.1 Description

Dans Gym, tous les environnements sont modélisés par la classe nommée `Env`. La fonction `gym.make` permet d'utiliser les environnements directement disponibles dans Gym en lui donnant en paramètre le nom de l'environnement à créer (par exemple `'CartPole-v1'` que nous verrons plus tard). Gym permet aussi de créer ses propres environnements en héritant de la classe `Env`, mais nous nous limiterons à ceux qui existent déjà.

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

La classe `Env` dispose de différents attributs et méthodes permettant aux agents d'interagir avec leur environnement, de l'observer et de réaliser des actions.

- **Attributs**
  - `observation_space` : de type `Space`, représente l'espace des observations (états) possibles
  - `action_space` : de type `Space` aussi, représente l'ensemble des actions que l'agent peut réaliser
  - `reward_range` : couple des valeurs de récompense minimale et maximale

- **Méthodes**
  - `reset()` : remet l'environnement dans son état initial et retourne l'observation correspondante.
  - `step(action)` : exécute l'action passée en paramètre et retourne l'observation de l'état courant de l'environnement, la récompense obtenue ainsi qu'une indication pour savoir si on a atteint un état terminal ou pas.
  - `render()` : affiche l'environnement selon le `render_mode` spécifié lors de sa création. Par exemple le mode `"human"` permet un affichage dans une fenêtre graphique.

```
env = gym.make("CartPole-v1", render_mode="human")
env.render()
```

Ces descriptions sont données à titre explicatif. Pendant le TP, il sera nécessaire de se référer à la documentation pour les différents détails d'implémentation (paramètres des fonctions, valeurs de retour etc.).

1.2.2 L'environnement Cart Pole

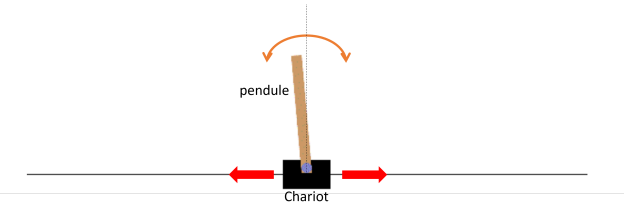

Figure 2: Le problème du Cart Pole

Le problème du Cart Pole est un problème dans le quel on doit maintenir en équilibre un pendule inversé accroché à un chariot qui peut se déplacer de gauche à droite sur un axe horizontal (illustration figure 2). Nous allons voir dans cette partie comment est définie l'environnement Cart Pole dans Gym (observations, actions, récompenses, états terminaux etc.).

1. Compréhension de l'environnement :
   (a) Créer une instance de l'environnement `"CartPole-v1"` sans spécifier le mode de rendu.
   (b) Observer les `Space` utilisés pour représenter les observations et les actions en les affichant sur la console.
   (c) Pour savoir à quoi correspondent ces spaces, rendez-vous sur la page de documentation de Cart Pole
   (d) Quelle est la valeur de la récompense pour chaque action que l'agent peut réaliser ?
   (e) Quels sont les états terminaux ?
2. Regardons maintenant comment exécuter une action.
   (a) Tout d'abord, générer une observation initiale en utilisant la fonction `env.reset` (sans spécifier d'argument).
   (b) Choisir au hasard une action `a` dans l'`action_space` en utilisant la fonction `env.action_space.sample`.
   (c) Utiliser la commande `env.step(a)` pour exécuter l'action `a` et observer les valeurs de retour de cette fonction en les affichant sur la console. Essayer de comprendre les différents termes retournés en vous référant à la documentation de la fonction step.

3. Maintenant qu'on a compris comment est définie l'environnement et comment réaliser une action, essayons de réaliser un épisode en entier. Pour rappel un épisode consiste à réaliser une succession d'actions, à partir d'un état initial, jusqu'à atteindre un état terminal.

   Écrire une fonction `do_episode` qui prend en paramètre un environnement, réalise un épisode entier et retourne la récompense totale obtenue sur l'épisode. Etant donné que l'on a pas encore de politique pour choisir l'action à réaliser, le choix pourra se faire aléatoirement comme pour la question précédente.

4. Pour observer le résultat graphiquement, créer un nouvel environnement `"CartPole-v1"` en mode rendu `"human"`. Appeler la fonction `do_episode` en lui passant l'environnement nouvellement créé en paramètre et afficher sur la console la récompense totale obtenue. Rajouter la commande `env.close()` pour arrêter l'affichage graphique de l'environnement à la fin de l'épisode.

   Vous devriez voir une fenêtre graphique s'ouvrir, mais très brièvement. Pour que la fenêtre reste plus longtemps, essayer de faire plusieurs épisodes (10 par exemple). n'oubliez pas de rajouter `env.close()` à la fin de votre code.

2 Implémentation du Q-learning

Actuellement on choisit des actions au hasard. Dans cette partie nous allons essayer de trouver une politique plus intelligente basée sur le Q-learning.

2.1 Discrétisation et création de la Q-table

La première étape est la définition de la Q-table et son initialisation. Cette dernière est un tableau dont les lignes représentent les états et les colonnes représentent les actions.

La Q-table étant basée sur un nombre fini d'états, il est nécessaire de discrétiser l'espace des observations. Pour cela on va découper chaque variable de l'`observation_space` en un nombre fini de segments.

1. (a) Créer une variable `cardinals` qui contient pour chaque variable de l'`observation_space`, le nombre de segments que vous aurez choisi pour discrétiser la variable. Environ 10 segments par variable devraient suffire.

   (b) Créer une variable `bounds` qui est une liste qui contient les plages de valeurs, sous forme de couples `(min, max)`, de toutes les variables de l'`observation_space`. Pour certaines variables, la plage de valeur est directement indiquée dans la documentation. Pour les autres variables, créer un nouvel environnement sans spécifier le mode de rendu et faire une boucle de plusieurs épisodes (≈ 1000) avec des actions aléatoires et récupérer les valeurs minimales et maximales observées.

2. (a) Écrire une fonction `discretize_value` qui prend 3 paramètres :
   - `x` : un nombre réel
   - `bound` : l'intervalle auquel appartient ce nombre sous forme d'un tuple `(a, b)`
   - `n` : le nombre de segments utilisés pour découper l'intervalle

   La fonction doit retourner, le numéro du segment auquel appartient x (entre `0` et `n-1`). De plus si `x <= a`, on retourne `0`. Et si `x >= b` on retourne `n-1`

   Exemple pour `x = 0.75`, `bound = (0, 2)` et `n = 4`, la fonction devrait retourner `1` comme illustré sur l'exemple ci-dessous.

   Indication : Pour tous les `x` dans l'intervalle `[a, b[`, la valeur de retour correspond exactement au nombre de segments entiers qui séparent `x` et `a`. L'opérateur `"//"` de python permet de faire une division entière.

   (b) Écrire une fonction `discretize_observation` qui permet de discrétiser une liste de nombre réels. La fonction prend 3 paramètres :
   - `observation` : une liste de nombre réels
   - `bounds` : liste des intervalles d'appartenance pour chaque valeur dans `observation`

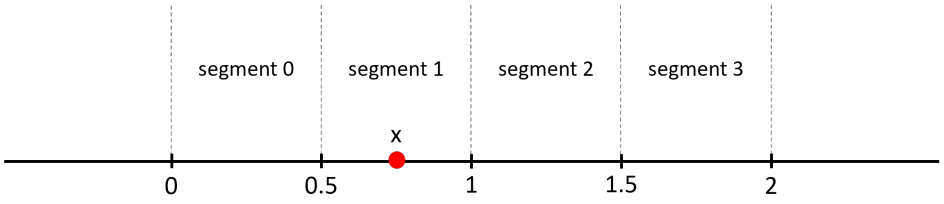

Figure 3: Illustration de la fonction de discrétisation

- `cardinals` : liste des nombres de segments utilisés pour découper chaque intervalle dans `bounds`

La fonction doit retourner une liste, `l`, telle que `l[i]` est le numéro du segment auquel appartient `observation[i]`. C'est à dire :

`l[i] = discretize_value(observation[i], bounds[i], cardinals[i])`

(c) On est maintenant capable de discrétiser notre espace d'observation en un nombre fini de vecteurs. par exemple pour `cardinals=[10, 10, 10, 10]`, on a 10000 vecteurs possibles. Ce qui correspond à 10000 états (lignes) pour notre Q-table. Pour passer d'un vecteur `v` à son état (et donc à la ligne correspondante dans la Q-table), utilisez la fonction `discrete_vec_to_state(vec, cardinals)` fournie dans le ficher `resources.py`.

Écrire maintenant une fonction `observation_to_state` permettant de passer directement d'une observation à son état en combinant les fonctions que l'on vient de voir.

3. Créer la Q-table (`Q`) en initialisant toutes ses valeurs à zéros. Pour cela on utilisera la fonction `np.zeros` qui permet de créer un tableau rempli de zéros en spécifiant la taille voulue.

```
>>> t = np.zeros((2, 3)) # pour créer un tableau de 2 lignes 3 colonnes
>>> t
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Le nombre de lignes (donc d'états) dépendra du nombre de segments que vous aurez choisi pour le découpage de chaque variable de l'`observation_space`.

2.2 Mise à jour de la Q-table

Une fois la Q-table initialisée, on peut commencer l'entraînement. Mais avant il faut définir une politique pour le choix des actions.

1. **Choix des actions : Exploration - Exploitation**

Au début de l'entraînement, étant donné que l'agent n'a pas beaucoup d'expérience, il choisit les actions à réaliser de façon totalement aléatoire (Exploration). Ensuite, la probabilité de choisir une action aléatoire diminue au fur et à mesure de l'apprentissage et l'agent exploite de plus en plus ses connaissances en se basant sur la Q-table (Exploitation). Nous faisons décroître cette probabilité exponentiellement en fonction du nombre d'épisodes déjà réalisés.

$$p = e^{-\lambda \cdot num} \tag{1}$$

Où $\lambda \ll 1$ est un réel positif et num est le numéro de l'épisode en cours. L'allure de la fonction est donnée sur la figure 4.

(a) Écrire une fonction `explore` qui permet de décider s'il faut faire de l'exploration ou pas. Cette fonction prendra en paramètre le numéro de l'épisode en cours (num) et un réel positif λ. La fonction doit retourner `True` (oui on explore) avec une probabilité $p = e^{-\lambda \cdot num}$ ou `False` (non on n'explore pas) avec une probabilité $1 - p$. Pour cela, sachant qu'un nombre réel tiré au hasard entre

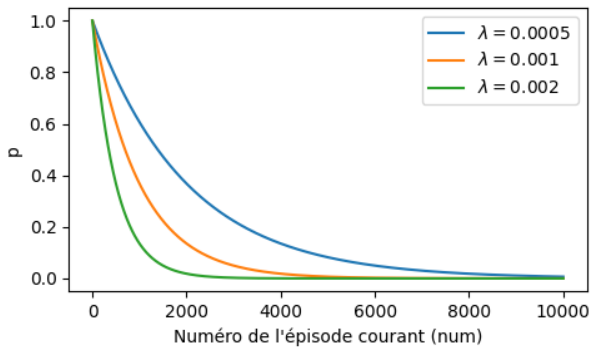

Figure 4: Allure de la décroissance de la probabilité

0 et 1 à une une probabilité p d'être inférieure à p (avec $p \in [0, 1]$), il suffira de tirer un nombre entre 0 et 1 an utilisant la fonction `np.random.rand`:

- Si ce nombre est inférieure à $p = e^{-\lambda \cdot num}$, on retourne `True`.
- Sinon, on retourne `False` : on n'explore pas (donc on exploite les connaissances de l'agent).

On utilisera la commande `np.exp(x)` qui permet de calculer l'exponentielle d'une valeur `x` passée en paramètre.

(b) Écrire une fonction `select_random_action` qui prend en paramètre le nombre d'actions (`n`) et retourne une action au hasard entre `0` et `n-1` en utilisant la fonction `np.random.randint`

(c) Écrire une fonction `select_best_action` qui prend en paramètre l'état courant et la Q-table et retourne l'action préconisée par la Q-table, c'est à dire celle avec le gain potentiel maximal pour l'état courant.

Indication : si `s` est l'état courant, `Q[s]` contient la liste des valeurs de la s-ième ligne de `Q`, c'est à dire les gains potentiels associés à chaque action depuis l'état `s`. Pour trouver l'action avec le gain potentiel maximal, on pourra utiliser la fonction `np.argmax` qui prend en paramètre une liste de valeurs et retourne la position de la valeur maximale.

(d) Finalement en combinant les trois questions précédentes, écrire une fonction `select_action` qui prend en paramètre l'état en cours, la Q-table, le numéro de l'épisode en cours (num) et λ, choisit entre exploration et exploitation selon les probabilités indiquées plus haut, puis retourne :

- une action aléatoire si exploration
- l'action préconisée par la Q-table si exploitation.

2. **Boucle de mise-à-jour**

Maintenant qu'on a une politique pour choisir les actions, on peut commencer l'entraînement à proprement parler. Celui-ci consiste simplement en une boucle de plusieurs épisodes au cours desquels la Q-table est mise-à-jour au fur et à mesure.

Créer un nouvel environnement sans spécifier le mode de rendu. Faire une boucle de plusieurs épisodes (≈ 10000) en utilisant la fonction `select_action` pour choisir les actions à réaliser (on pourra prendre $\lambda = 1e-3$) et en enregistrant dans une liste les récompenses totales obtenues après chaque épisode.

Après chaque action a depuis un état s, mettre à jour la Q-table selon l'équation d'optimalité de Bellman.

$$Q[s, a] = \alpha \left(r + \gamma \cdot max_{a'} Q[s', a']\right) + (1 - \alpha) Q[s, a] \tag{2}$$

Où $\alpha \in [0, 1]$ est le facteur d'apprentissage, $\gamma \in [0, 1]$ est le facteur d'actualisation, r est la récompense obtenue après avoir réalisé l'action a et s' est le nouvel état. On pourra choisir $\alpha = 0.1$ et $\gamma = 0.98$.

Syntaxe python :

```
Q[s, a] = alpha*(r + gamma*max(Q[new_s])) + (1-alpha)*Q[s, a]
```

3. Utiliser la fonction `show_rewards` dans `resources.py` pour afficher l'évolution moyenne des récompenses au cour de l'entraînement. Elle prend en paramètre la liste des récompenses obtenues pour chaque épisode au cours de l'entraînement.

2.3 Test de la Q-table obtenue

1. Créer u fonction `do_episode` en rajoutant la Q-table en paramètre et en choisissant l'action à réaliser en fonction de la Q-table.

2. Créer un nouvel environnement en mode `"human"` et regarder sur quelque épisode comment se comporte l'agent.