École national Supérieure d'Informatique
2ième année cycle supérieur (2023-2024)
Système intelligents et données (SID)
Machine Learning

Workshop 03. Q-learning

Il existe deux types d'apprentissage par renforcement selon le type d'interaction :

- **Online** : l'agent collecte directement des données : il collecte un lot d'expériences en interagissant avec l'environnement. Ensuite, il utilise cette expérience immédiatement (ou via un tampon de relecture) pour en tirer des leçons (mettre à jour sa politique).
- **Offline** : l'agent utilise uniquement les données collectées auprès d'autres agents ou de démonstrations humaines. Il n'interagit pas avec l'environnement.

Dans ce workshop, nous allons implémenter un agent générique qui utilise une table Q. Aussi, nous allons simuler l'environnement taxi-passager qui utilise des interactions online.

1 Agent

Ici, nous commençons par réaliser les fonctions de l'agent. Nous allons implémenter Q-learning où l'agent possède une matrice des états et des actions.

1.1 Création de la table Q

Étant donné n états et m actions, nous devons créer une matrice $Q[n, m]$ initialisée à 0. Dans ce cas, implémenter une fonction **creer_Q** qui retourne cette matrice :

```
def creer_Q(nbr_etats: int, nbr_actions: int) -> np.ndarray:
```

1.2 Choix de l'action à prendre

L'agent doit choisir une action qui est définie par un entier entre 0 et m (nombre des actions, aussi colonnes de Q). Il y a deux méthodes pour choisir l'action : l'exporation et l'exploitation.

Dans l'exploration, le numéro de l'action est choisi aléatoirement parmi les actions de la table Q. Donc, l'agent va choisir une action (qui peut ne pas être la meilleure) afin de tester/explorer son environnement.

```
def exploration(Q: np.ndarray) -> int:
```

Dans l'exploitation, l'action choisie est celle qui a une valeur max parmi celles de l'état courant. Donc, l'agent va exploiter le chemin optimal afin d'arriver au but final.

```
def exploitation(Q: np.ndarray, etat: int) -> int:
```

Le choix de l'action à prendre dépend d'une probabilité ϵ d'exploration. Sinon, l'action doit être choisie par l'exploitation.

```
def choisir_action(Q: np.ndarray, etat: int, epsilon: float=0.2) -> int:
```

20% Explor et autres exploita

1.3 Mise à jours de la table Q

Étant donné une table Q avec :

- s_t, a_t : état courant et action courante
- α : taux d'appentissage
- r : récompense de l'environnement
- γ : facteur d'actualisation
- s_{t+1} : l'état suivant

La mise à jour d'une cellule $Q(s_t, a_t)$ par l'équation 1.

on garde alpha *la probabilité de l'action et on ajoute 1-alpha de nouvelle proba

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (r + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{1}$$

La fonction prend en argument : la table Q, l'état courant, le nouveau état, l'action courante, le taux d'apprentissage, la récompense et le facteur d'actualisation. Elle retourne la nouvelle table mise à jour.

```python
def mettre_ajour_Q(Q: np.ndarray, etat: int, netat: int, action: int, alpha: float, r: float, gamma: float) -> np.ndarray:
```

1.4 Classe Agent

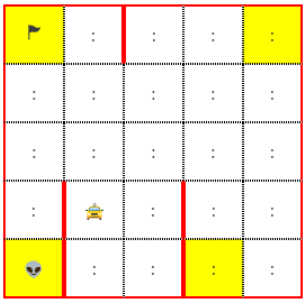
La classe **Agent** est déjà implémentée. Elle définit :

- Un constructeur avec le nombre des états, le nombre des actions possibles, taux d'apprentissage et la probabilité d'exploration.
- **set_etat** : pour mettre à jour l'état de l'agent
- **choisir_action** : pour récupérer l'action suivante de l'agent
- **appliquer** : pour appliquer la récompense

```python
class Agent:
    def __init__(self, nbr_etats: int, nbr_actions: int, alpha: float, epsilon=0.2):
        ...

    def set_etat(self, etat: int):
        ...

    def choisir_action(self) --> int:
        ...

    def appliquer(self, netat: int, action: int, r: float, gamma: float):
        ...
```

2 Environnement

Ici, nous allons implémenter le problème du taxi et passager (voir la figure 1). Notre environnement est un espace divisé en nb_l lignes et nb_c colonnes pour indiquer la position. Il contient, aussi, un nombre d'arrêts nb_a (en jaune). La position du taxi est encodée en utilisant le numéro de la ligne et de la colonne (un tuple des coordonnées). La destination est le numéro de l'arrêt ($0 <= dst < nb_a$). La position du passager est représentée par le numéro de l'arrêt ($0 <= psg < nb_a$) plus un numéro $psg = nb_a$ indiquant que le passager est à l'intérieur du taxi.



emplacement taxi * emplacement passager
5*5:nbrL*nbrC
4:arrets , 1:dans taxi
4:destinations
5*5*(4+1)*4 =nbr etats 500

FIGURE 1 : Environnement taxi

2.1 États

Étant donné :

- un espace divisé en nb_l lignes et nb_c colonnes, donc nous avons $nb_l * nb_c$ positions possibles de la voiture
- un nombre d'arrêts nb_a, donc nous avons $nb_a + 1$ positions du passager (soit dans un de ces arrêts, soit dans la voiture)
- une destination dst, qui est un des arrêts ($0 <= dst < nb_a$), donc nous avons nb_a destinations possibles

Donc, au total, nous aurons $nb_l * nb_c * (nb_a + 1) * nb_a$ états possibles. Nous allons organiser les états comme une liste de nb_l listes de nb_c listes de $(nb_a + 1)$ listes de nb_a éléments. En quelque sorte, c'est un tenseur (une matrice multidimensionnelle). Le numéro de l'etat est simplement son ordre dans cette structure en appliquant une opération "Flatten".

Un état est un numéro qui peut être calculé en se basant sur la position du taxi, le numéro de l'arrêt de démarrage, le numéro de l'arrêt destinataire, le nombre des colonnes, des lignes et des arrêts. Une fonction qui calcule le numéro d'état est fournie.

```
1 ||        def encoder_etat(pos: Tuple[int, int], psg: int, dst: int, nb_l: int, nb_c: int, nb_a: int) -> int:
```

La fonction inverse calcule le numéro de ligne, le numéro de la colonne, la position du passager et la position d'arrêt. Elle se base sur le numéro d'état, nombre des lignes, nombre des colonnes et le nombre des arrêts.

```
1 ||        def decoder_etat(etat: int, nb_l:int, nb_c: int, nb_a: int) -> Tuple[int, int, int, int]:
```

vecteur 4 elements :Ligne , colonne dans la ligne 5, emplacement passager dans la ligne 4+1, emplacement de destination 4
qd flatten this vect on obtient i=un numero unique de l'etat

2.2 Actions

Il existe 6 actions possibles :

- 4 actions de repositionnement : gauche (0), droit (1), avant (2) et arrière (3)
- une action pour prendre le passager (4)
- et une autre pour le déposer (5)

En se basant sur l'action et la position, la fonction de repositionnement retourne la nouvelle position. Elle ne prend pas en considération les contraintes comme les barrières (donc, il faut vérifier les barrières avant d'appeler cette fonction).

```
1 || def repositionner(pos: Tuple[int, int], action: int) -> Tuple[int, int]:
```

2.3 Récompense

La fonction de récompense a comme entrée :

- etat : l'état courant de l'agent
- action : numéro de l'action choisie par l'agent
- nb_l, nb_c : nombre des lignes et des colonnes dans l'environnement
- arrets : une liste des positions des arrets. La position est encodée sous forme d'un tuple (x, y). P.S. Les tuples sont hashables ; donc on peut vérifier leur existence dans la liste en utilisant l'opérateur "in".
- bar : un dictionnaire pos : list entiers. Si une position existe, on aura une liste des actions interdites (actions de positionnement).

Elle doit retourner : la récompense, l'état suivant et un booléen qui indique la fin.

```python
def calculer_recompense(etat: int, action: int,
                        nb_l:int, nb_c: int,
                        arrets: List[Tuple[int, int]],
                        bar: Dict[Tuple[int, int], Set[int]]) -> Tuple[float, int, bool]:
```

La récompense est calculée comme suit :

- Pour chaque action exécutée, une récompense de -1 est attribuée pour minimiser nbr d'actions
- Si l'agent essaye de déposer ou prendre un passager illégalement, une récompense de -10 est attribuée en plus. L'action "déposer" est considérée illégale si le passager n'est pas dans le taxi ou si le lieu de dépôt n'est pas l'arrêt destinataire. L'action "prendre" est considérée illégale si le passager n'est pas dans la position actuelle ou il est déjà dans la voiture.
- Si l'agent dépose le passager dans l'arrêt destinataire, il aura une récompense de +20 en plus.

L'état suivant est calculé comme suit :

- L'état suivant est celui actuel sauf dans les cas suivants
- Si le passager monte dans le taxi, l'index du passager sera "nb_a" (dans la voiture).
- Si le passager arrive à sa destination en sortant du taxi, l'index du passager sera "dst". La fin sera True.
- Dans le cas d'une action de positionnement (0, 1, 2, 3), si la position n'est pas dans celles du barrière "bar" ou elle existe mais l'action n'existe pas dans la liste des actions interdites, la nouvelle position et le nouveau état sont calculés.

2.4 Classe Environnement

La classe **Environnement** est déjà implémentée. Elle définit :

- Un constructeur avec le nombre des lignes, le nombre des colonnes, les positions des arrêts, les barres qui est un dictionnaire des positions comme clés et un ensemble des actions interdites comme valeur, et le facteur d'actualisation.
- **ajouter_agent** : pour créer un agent en se basant sur le taux d'apprentissage et la probabilité d'exploration.
- **encoder_etat** : pour encoder l'état (voir la sous-section : États)
- **decoder_etat** : pour décoder l'état (voir la sous-section : États)
- **initialiser** : pour initialiser l'environnement en se basant sur la position initiale de la voiture, l'arrêt initial du passager et l'arrêt destinataire.

- **transporter** : pour exécuter l'action de transporter le passager vers sa destination. Cette fonction retourne l'historique de la position de la voiture, du passager, destination, action, récompense et est-ce qu'on a atteint la destination. Aussi, cette fonction peut dessiner une animation en temps réel.
- **dessiner** : dessiner l'environnement avec l'état actuel

```python
class TaxiEnv():
    def __init__(self, nb_l:int, nb_c: int, arrets: List[Tuple[int, int]],
                       bar: Dict[Tuple[int, int], Set[int]], gamma: float = 0.5):
        ...

    def ajouter_agent(self, alpha: float, epsilon=0.2):
        ...

    def encoder_etat(self, pos: Tuple[int, int], psg: int, dst: int):
        ...

    def decoder_etat(self, etat: int) -> Tuple[int, int, int]:
        ...

    def initialiser(self, pos: Tuple[int, int], psg: int, dst: int):
        ...

    def transporter(self, plot=False):
        ...

    def dessiner(self):
        ...
```

Liens et Ressources

Voici quelques outils pour tester l'apprentissage par renforcement :

- OpenAI Baselines : `https://github.com/openai/baselines`
- Intel Coach : `https://github.com/IntelLabs/coach`
- Stable Baselines : `https://github.com/DLR-RM/stable-baselines3`
- TF-Agents : `https://github.com/tensorflow/agents`
- Keras-RL : `https://github.com/keras-rl/keras-rl`
- Tensorforce : `https://github.com/tensorforce/tensorforce`
- Chainer RL : `https://github.com/chainer/chainerrl`
- Mushroom RL : `https://github.com/MushroomRL/mushroom-rl`
- Acme : `https://github.com/deepmind/acme`
- Dopamine : `https://github.com/google/dopamine`
- RAY : `https://github.com/ray-project/ray`

Environnements :

- Gym : `https://gym.openai.com/`
- iGibson : `http://svl.stanford.edu/igibson/`