Introduction à l'apprentissage par renforcement

Culture Sciences de l'Ingénieur

Édité le 20/07/2022

école normale supérieure paris-saclay

Anthony JUTON – Valentin NOËL – Rida LALI

Cette ressource est issue d'une publication du numéro 109 de La Revue 3EI de juillet 2022 et fait partie du « Dossier Intelligence Artificielle » [11] sur Culture Sciences de l'Ingénieur. *Anthony Juton en professeur agrégé au DER Nikola Tesla de l'ENS Paris-Saclay, Valentin Noël est doctorant en imagerie médicale au Laboratoire SATIE, et Rida Lali est étudiant au DER d'informatique de l'ENS Paris-Saclay.*

Cette ressource présente une méthode d'apprentissage de l'intelligence artificielle, bien adaptée à des problèmes pour lesquels il est possible de simuler le comportement du système dans son environnement (jeux vidéo, conduite autonome, asservissement de systèmes mécaniques...). Les données étant générées par l'interaction entre le système et son environnement, il n'est pas nécessaire de disposer d'un jeu de données comme pour l'apprentissage supervisé.

Après avoir présenté l'apprentissage par renforcement, la ressource développe en détail 2 algorithmes : le Q-Learning et le Deep Q-learning. S'intéressant au principe de l'apprentissage, il ne demande pas des connaissances en python très importantes.

Les codes commentés sont fournis sur Culture Sciences de l'Ingénieur [10] et sont un bon support pour qui voudrait adapter son propre système pour y faire de l'apprentissage par renforcement.

1 – Introduction

L'apprentissage par renforcement (reinforcement learning RL) est une méthode d'apprentissage qui s'intéresse à la prise de décision.

Depuis l'état s (s comme state) de l'environnement, l'agent utilise une politique π pour choisir une action a. L'apprentissage par renforcement vise à optimiser la politique d'action π de l'agent grâce à un jeu de récompenses positives et négatives.

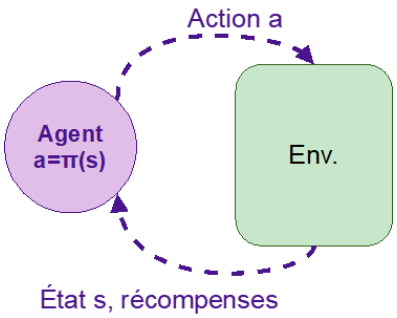

Figure 1 : Schéma d'interaction entre l'agent et son environnement

L'apprentissage demandant un grand nombre d'expérimentation, il est très utile voir indispensable de disposer d'un environnement simulé. Les jeux vidéo étant un environnement simulé avec des récompenses (le score), ils sont un support classique pour l'apprentissage par renforcement.

Le logiciel MuZero de DeepMind (société rachetée par Google), successeur d'Alpha Go, montre depuis 2018 le potentiel de l'apprentissage par renforcement, sans jeu de données préalable, pour être le meilleur au jeu de Go mais aussi à de nombreux jeux vidéo. [1]

Cette ressource, non exhaustive sur le sujet, présentera l'environnement des process markoviens dans lequel se place l'apprentissage par renforcement. Ensuite, il détaillera un des algorithmes d'apprentissage les plus simples : l'apprentissage tabulaire Q-learning. Enfin, nous introduirons l'apprentissage par renforcement profond où la politique d'action est issue d'un réseau de neurones, là encore à partir d'un des algorithmes les plus simples, le Deep Q-learning.

Trois autres articles du « Dossier Intelligence Artificielle » [11] fournissent des applications pour pratiquer les outils et algorithmes avancés de l'apprentissage par renforcement.

- « Stabilisation d'un pendule inversé à l'aide d'un apprentissage par renforcement » [12] utilise Matlab ;
- « Introduction aux bibliothèques Gym et Stable Baselines pour l'apprentissage par renforcement » [13] utilise python et les très populaires bibliothèques gym et stable baselines.
- « Apprentissage par renforcement de la conduite d'un véhicule sur AirSim » [14] utilise python, gym, stable baselines et le simulateur de voitures autonomes réaliste AirSim.

2 – L'environnement

Avant de manipuler et de formaliser l'apprentissage par renforcement, il faut savoir identifier un problème adapté à l'apprentissage par renforcement pour sa résolution.

Comme décrit dans l'introduction, l'apprentissage par renforcement s'appuie sur l'interaction entre l'agent et son environnement.

L'**environnement** est d'abord dans un **état** s_t.

L'**agent** effectue une **action** a_t qui va avoir un impact sur l'environnement.

L'état de l'environnement est alors modifié (ou pas), il en résulte un nouvel état s_{t+1} ainsi qu'une **récompense** r_t qui permettra d'évaluer la pertinence de l'action choisie.

La fonction qui établit le passage de l'environnement de l'état s_t à l'état s_{t+1} sous l'effet de l'action a_t est nommée **fonction de transition**.

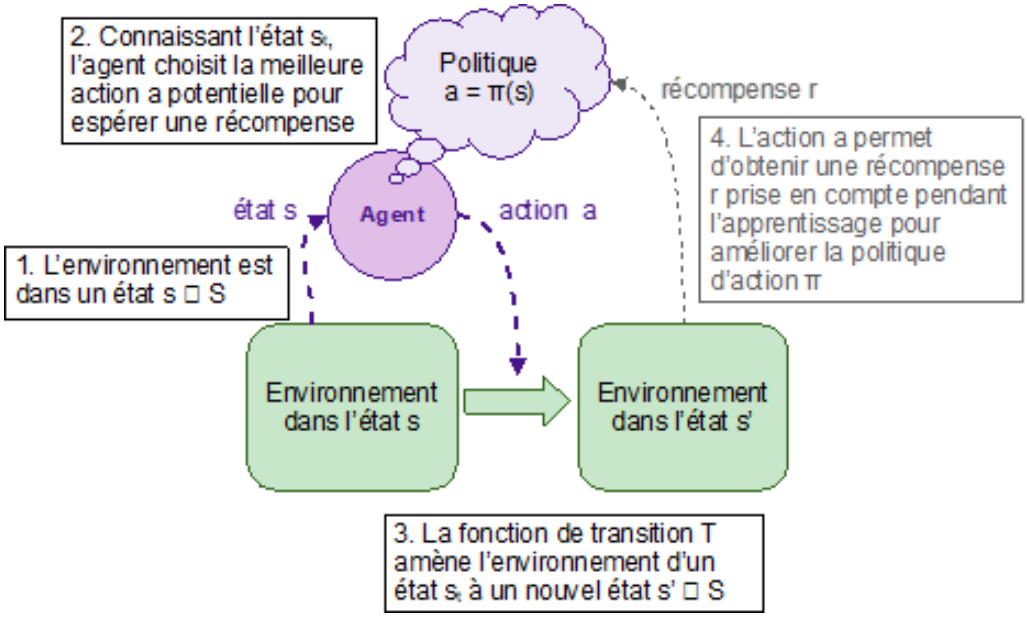

Figure 2 : Principe de l'apprentissage par renforcement

L'optimisation de la politique ne se fait que pendant les périodes d'apprentissage. Une fois cet apprentissage jugé correct, il est possible d'utiliser la politique π pour agir sur l'environnement, sans chercher à l'améliorer.

2.1 – Le processus de décision markovien

L'apprentissage par renforcement se place dans le cadre des processus de décision markovien (Markov Decision Process MDP). L'évolution de l'environnement doit respecter la propriété de Markov : la fonction de transition T(s,a,s') doit être une fonction stochastique attribuant, pour une action a exécutée dans un état s, les probabilités de se retrouver dans chaque état s' et cette distribution de probabilité doit être indépendante des actions et état précédents.

Ainsi, le processus est modélisé par le quadruplet {S, A, T, R}

- **un ensemble d'états S**. ensemble (continu ou discret) des états de l'environnement que perçoit l'agent ;
- **un ensemble d'actions A**, discret ou continu ;
- **une fonction de transition T(s,a,s')** représentant la probabilité de se retrouver dans l'état s' en effectuant l'action a depuis l'état s ;
- **une fonction de récompense R(s,a,s')**, indiquant la récompense obtenue lors du passage de s à s' en effectuant l'action a.

Notons que dans un **environnement déterministe**, la fonction de transition est simplifiée : elle indique directement l'état s' dans lequel se retrouve l'environnement après l'exécution de l'action a depuis l'état s.

Afin de mieux illustrer ce qu'est un processus de décision markovien, voici quelques exemples de problèmes adaptés à l'apprentissage par renforcement

2.2 – Exemple d'environnement

2.2.1 – Le pendule inversé

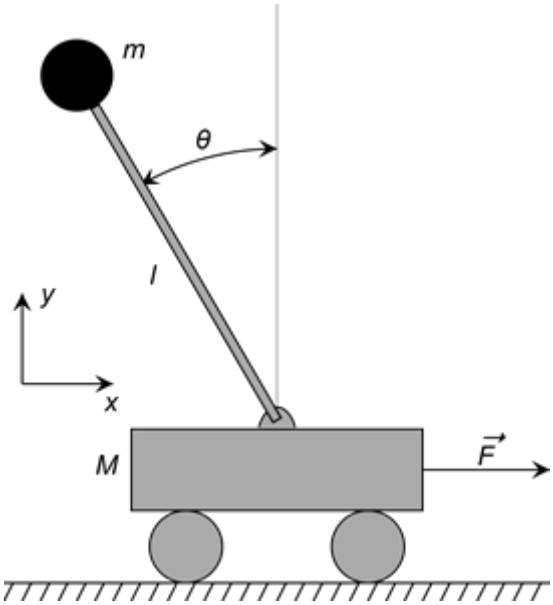

Figure 3 : Schéma d'un pendule inversé

Le pendule inversé est un problème classique de contrôle optimal dont l'objectif est de maintenir à l'équilibre un poids sur un chariot mobile.

L'état est décrit par un vecteur composé de l'angle θ, de la vitesse angulaire ω, de la position relative du chariot ainsi que de la vitesse de ce dernier. L'ensemble S des états est donc un ensemble continu, que l'on peut si besoin discrétiser.

Les actions correspondent aux différentes valeurs de la force appliquée au chariot. C'est là encore un ensemble continu que l'on peut si besoin discrétiser.

En l'absence de frottement sec et si on se limite à de petits angles du pendule, la fonction de transition peut être modélisée par des équations linéaires indiquant la valeur suivante de l'état en fonction de l'état actuel et de l'action effectuée. On obtient alors un problème d'automatique classique. En présence de frottement sec et si on considère de plus larges déplacements angulaires du pendule, le problème est beaucoup plus complexe, mais reste déterministe. On pourrait alors écrire la **fonction de transition**, non linéaire.

Pour la **fonction de récompense**, on peut imaginer donner une récompense positive liée à l'angle que fait le pendule avec la verticale et une récompense négative en cas de chute du pendule (angle supérieur à 45° par exemple)

La ressource « *Stabilisation d'un pendule inversé à l'aide d'un apprentissage par renforcement* » [12] propose un exemple de séances de travaux pratiques autour de ce problème avec Matlab et un passage de la simulation à la réalité avec le système ControlX. La ressource « *Introduction aux bibliothèques Gym et Stable Baselines pour l'apprentissage par renforcement* » [13] propose d'utiliser la bibliothèque Gym pour l'apprentissage d'un pendule inversé (Cartpole).

2.2.2 – Corps articulé

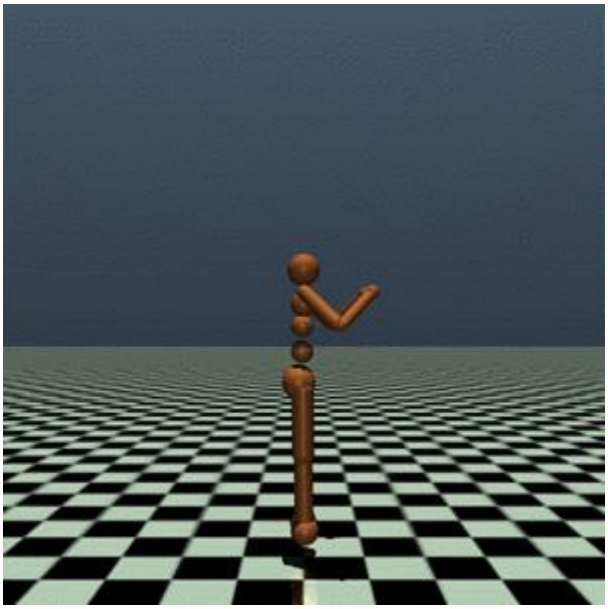

Figure 4 : Simulation d'un humanoïde dans l'environnement Gym MuJoCo.
Image extraite de la documentation de Gym

Gym propose plusieurs problèmes alliant robotique, biomécanique et environnement 3D. Le but de celui présenté en**Erreur ! Source du renvoi introuvable.**, l'un des plus ambitieux, est de maintenir un squelette mécanique debout et de le faire se déplacer sans le déséquilibrer.

L'état est constitué de l'angle et de la position relative de chaque articulation.

L'action est l'ensemble des couples appliqués à chaque articulation.

La fonction de transition peut prendre en compte les frottements secs de chaque articulation et donc être non linéaire. Elle peut aussi prendre en compte du bruit sur les commandes ou sur certains paramètres et des glissements stochastiques. Elle devient donc probabiliste.

La récompense est donnée par la combinaison des deux objectifs : être debout et se déplacer vers l'avant.

Un système physique réel étant infiniment complexe, pour passer du système réel au système simulé, il y a besoin d'expertise pour choisir les hypothèses que l'on peut faire pour simplifier le modèle sans être dans un cas trop simpliste. Ce choix est d'autant plus important, si on souhaite appliquer la politique apprise en simulation sur le système réel ensuite.

En parallèle des problèmes d'ingénierie pour lesquels on dispose ou on développe un simulateur, les jeux vidéo (ou les jeux ayant une variante vidéo) sont particulièrement propices au développement d'une IA entraînée par apprentissage par renforcement.

2.2.3 - Jeux vidéo Atari

Les jeux vidéo individuels fournissent un environnement informatique bien adapté à l'apprentissage par renforcement.

Les états sont les valeurs des pixels de l'écran de quelques images successives, **les actions** sont les actions possibles pour le joueur, **la fonction de transition** est fournie par le jeu (l'évolution de l'affichage à chaque action du joueur, qui doit respecter les propriétés du process de Markov), **les récompenses** sont les variations de score à chaque action (ou non action).

On pourrait imaginer des états de plus haut niveau (position de la balle, des briques et de raquettes pour un la casse-brique breakout par exemple).

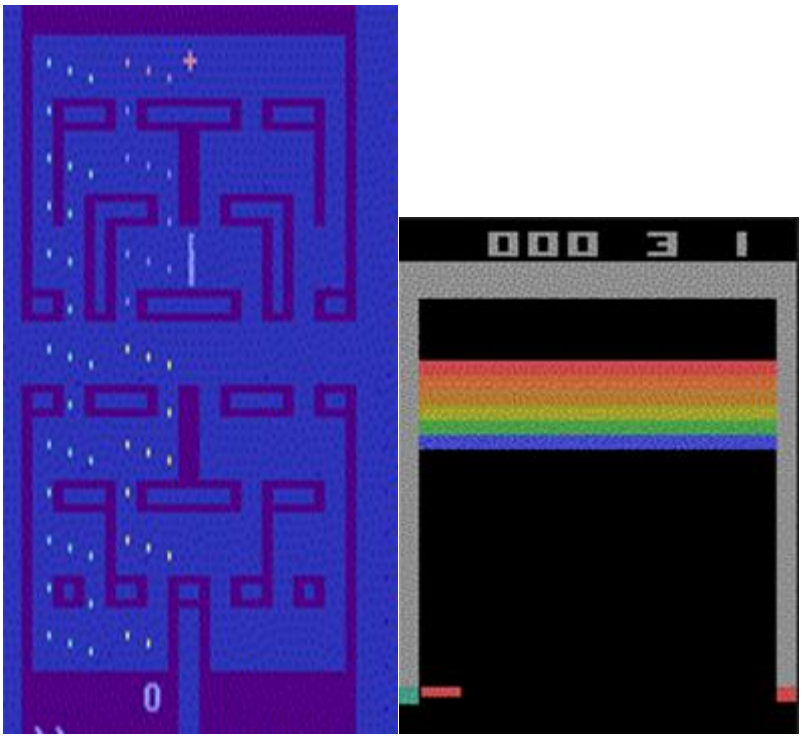

Figure 5 : Jeux Atari Alien et Breakout, simulés à l'aide de l'environnement Gym.
Images extraites de la documentation de Gym

L'environnement Gym offre la possibilité de simuler différents jeux afin d'y implémenter des modèles d'apprentissage par renforcement. [9]

2.2.4 - Jeux à deux joueurs (Pong, Morpion, Go, Echec, …)

Les jeux à deux joueurs, notamment les échecs et le go, demandent de simuler un adversaire pour l'apprentissage. Ils ont largement contribué aux succès médiatiques de l'IA : DeepBlue vainquit la champion Gary Kasparov aux échecs en 1997 et AlphaGo battit Lee Sedol au Go en 2017. Aucun joueur humain n'a aujourd'hui l'ambition de provoquer une nouvelle partie contre un de ces systèmes toujours plus performants.

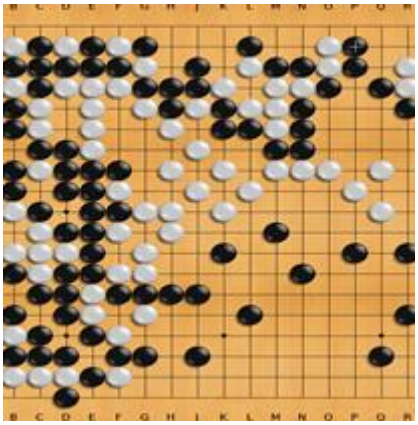

Figure 6 : Jeu de go

En 1997, l'algorithme de DeepBlue, sans réseau de neurones, reposait sur le parcours dans un immense arbre. La programmation de DeepBlue avait requis les services d'un champion d'échec et les données de nombreuses parties.

Le Jeu de Go est un jeu d'une complexité telle qu'il ne peut être modélisé par un simple arbre de possibilité. En 2017, AlphaGo utilisait un réseau de neurones pour trouver un chemin dans cet arbre. Le réseau de neurones était entraîné au début pour imiter les meilleurs joueurs de go, à partir de leurs parties enregistrées (on peut parler d'apprentissage par imitation). L'apprentissage par renforcement prenait alors la suite.

Depuis 2018, AlphaGo Zero, renommé AlphaZero, puis MuZero apprennent avec succès les jeux de Go, Echecs, Shogi, Atari, sans aucune donnée préalable, avec des algorithmes issus de l'apprentissage par renforcement, en faisant des parties contre d'autres instances d'eux-mêmes.

On peut alors imaginer le problème posé de la manière suivante : **les états** sont donnés par la position de toutes les pièces du jeu, **l'action** caractérise l'emplacement de la prochaine pièce, **la fonction de transition** est évidente, donnée par la règle du jeu, et **la récompense** est positive lorsque l'agent gagne la partie et négative s'il la perd.

Par exemple, il est possible en travaux pratiques avec des étudiants de modéliser un jeu de morpion (Tic-Tac-Toe en anglais). Ce jeu, bien plus simple que les jeux de Go ou d'échec, permettra de développer les premières intuitions concernant l'apprentissage à deux joueurs.

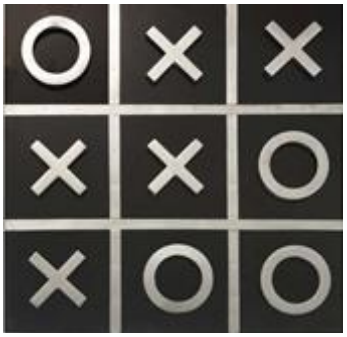

Figure 7 : jeu de morpion

- **Les états** sont le contenu des 9 cases (soit 3^9= 19683 états possibles, sans a priori ni simplification),
- **Les 9 actions possibles** sont la case jouée.
- **La fonction de transition** est évidente (on ajoute la case jouée à l'état du jeu), si ce n'est qu'il faut y inclure le jeu de l'adversaire (et donc la case jouée par l'adversaire).
- **La fonction de récompense** peut donner 0,1 point négatif pour une action interdite (action sur une case déjà jouée), 1 point pour une victoire et -1 pour une défaite.

Pour commencer l'apprentissage de la politique de l'agent sur le jeu à 2 joueurs, il faut donner un algorithme (soit hasard complet, soit un peu intelligent) à l'adversaire de cet agent.

Ces exemples de problèmes présentés, intéressons-nous maintenant à l'apprentissage de la politique d'action.

3 – L'apprentissage

Une fois créé l'environnement {S,A,T,R} respectant les propriétés des process de Markov, il s'agit de mettre en place l'apprentissage par l'agent de sa politique π d'action, en essayant d'approcher le plus possible de la politique optimale notée π^*.

Nous étudierons ici deux méthodes en détail, avec des exemples simples codés sans usage de framework, « from scratch » (les codes sont donnés sur [10]). La première, une méthode tabulaire nommée Q-learning, relativement simple, permettra de bien comprendre les principes de l'apprentissage par renforcement. Ensuite, nous étudierons une méthode d'apprentissage profond où la politique d'actions est un réseau de neurones, ce qui permettra d'introduire les méthodes plus avancées d'apprentissage utilisées dans les autres ressources sur l'apprentissage par renforcement.

4 – Apprentissage tabulaire Q-Learning

On s'intéresse ici au problème très classique du lac gelé (frozen lake – un esquimau cherche à rejoindre un igloo sur un lac gelé en évitant le trou), légèrement revisité pour tenir compte du changement climatique : un lapin cherche à rejoindre une carotte sur un pré glissant en évitant le renard.

A travers cet exemple, nous allons introduire les notations utilisées pour l'apprentissage par renforcement et l'algorithme de Q-learning utilisable pour les problèmes ayant un nombre limité d'états et d'actions. Il est possible de modifier le code pour l'utiliser pour résoudre un autre problème « markovien » à nombres d'états et actions limités.

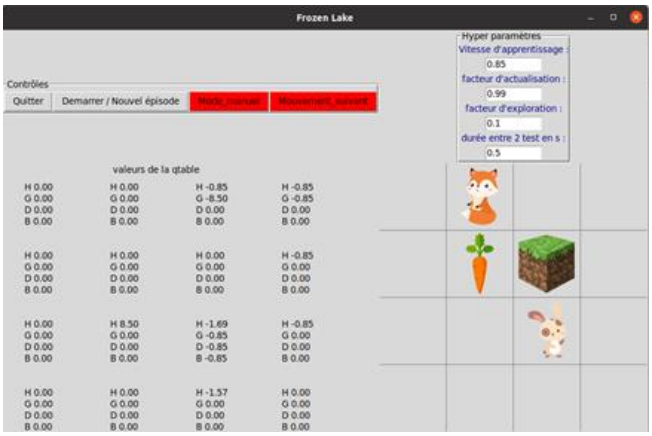

Figure 8 : Environnement Frozen Lake revisité

Le projet comporte 2 fichiers, disponibles sur le dépôt indiqué en [10] :

- FrozenLake_2022_fenetre.py contient la description des fenêtres, des affichages et des boutons. Il n'est pas utile de l'ouvrir pour ce TP.
- FrozenLake_2022.py contient la description de l'environnement et l'apprentissage. Il reprend le formalisme utilisé par la populaire bibliothèque d'apprentissage Gym, sans l'utiliser.

L'environnement décrit dans la classe *Game* comprend un lapin qui, à partir de la case départ, doit atteindre la carotte sans sortir de la grille et en évitant le renard. Les blocs ne peuvent être franchis. L'agent est le contrôle du lapin.

Une fois la grille créée, les bloc, carotte et renard sont fixes. **L'état** (state) de l'environnement se limite donc à la position du lapin.

```
def _get_state(self):
    return self._position_to_id(*self.position_lapin)
```

Les actions sont les commandes de déplacement envoyées par l'agent au lapin : Haut, Gauche, Droite, Bas.

```
#Définition de l'environnement, états, récompenses, actions
class Game:
    ACTION_H = 0
    ACTION_G = 1
    ACTION_D = 2
    ACTION_B = 3
    #table des actions possibles
    ACTIONS = [ACTION_H, ACTION_G, ACTION_D, ACTION_B]
    #table des noms des actions
    ACTION_NAMES = ["H", "G", "D", "B"]
    #coordonnées des mouvements
    MOUVEMENTS = {
        ACTION_H: (0, -1),
        ACTION_G: (-1, 0),
        ACTION_D: (1, 0),
        ACTION_B: (0, 1)
    }
    #nombre d'actions possibles
    num_actions = len(ACTIONS)
```

La fonction de transition renvoie la nouvelle position du lapin si le déplacement était possible et la position actuelle sinon. Dans l'exemple suivi ici, la fonction de transition est déterministe : si la commande est Haut, le lapin va à la case du dessus, sauf s'il n'y a pas de case ou si un bloc est sur cette case. Il est aussi possible d'activer la section en commentaire pour ajouter un peu d'aléatoire : l'herbe est alors mouillée et de façon aléatoire, le lapin glisse parfois et exécute alors une action proche de celle souhaitée.

La fonction de récompense attribue :

- -1 si la case de destination n'existe pas ou si elle est occupée par un bloc ou si elle est vide. On pénalise ainsi les déplacements inutiles.
- -10 si la case de destination est occupée par un renard.
- +10 si la case de destination est occupée par une carotte.
- Il est évidemment possible de développer une autre fonction de récompense, en ne pénalisant pas par exemple les déplacements inutiles.

La fonction move, appelée pour chaque action, comprend à la fois la fonction de transition (elle renvoie l'état suivant) et la fonction de récompense (elle renvoie la récompense) :

```python
#fonction de déplacement : recoit une action et renvoie :
#le nouvel état de la grille, la récompense,
#si c'est terminé et des infos (0 ici)
#Il est possible d'y ajouter un peu d'aléatoire (les glissades du lapin)
def move(self, action):
    self.counter += 1

    if action not in self.ACTIONS:
        raise Exception("Invalid action")

    #ajout d'une action aléatoire :
    #le lapin glisse et ne fait pas ce qu'il aurait prévu
    # choice = random.random()
    #if choice < 0.1 :
    #    action = (action + 1) % 4
    #elif choice < 2 * 0.1 :
    #    action = (action - 1) % 4

    #calcul du déplacement prévu -> new_x et new_y
    d_x, d_y = self.MOUVEMENTS[action]
    x, y = self.position_lapin
    new_x, new_y = x + d_x, y + d_y

    #calcul des conséquences de l'action :
    #nouvelle position, récompense, fin du jeu, info(inutilisée)
    if self.block == (new_x, new_y):
        return self._get_state(), -1, False, 0
    elif self.renard == (new_x, new_y):
        self.position_lapin = new_x, new_y
        return self._get_state(), -10, True, 0
    elif self.carotte == (new_x, new_y):
        self.position_lapin = new_x, new_y
        return self._get_state(), 10, True, 0
    elif new_x >= self.n or new_y >= self.m or new_x < 0 or new_y < 0:
        return self._get_state(), -1, False, 0
    elif self.counter > 190:
        self.position_lapin = new_x, new_y
        return self._get_state(), -10, True, 0
    else:
        self.position_lapin = new_x, new_y
        return self._get_state(), -1, False, 0
```

On notera que la classe *Game* comprend également une fonction *generate_game* pour générer un environnement de manière aléatoire (position des blocs, renard, carotte) et une fonction *reset* pour remettre le lapin à sa position initiale.

On s'intéresse désormais à l'apprentissage permettant à l'agent d'optimiser sa politique d'actions et ainsi au lapin de trouver son chemin vers la carotte.

4.1 - Principe du Q-Learning

4.1.1 - La Q-Table

Le nombre d'états étant limité (16 ici) et le nombre d'actions étant limité également (4), il est possible de modéliser la politique d'action π par une table de qualité (**Q-table**) associant à chaque couple état-action une valeur de « qualité ».

	Action HAUT	Action GAUCHE	Action DROIT	Action BAS
État (0,0)	0.1	5.0	7.1	2.2
État (0,1)	-0.1	5.0	0.1	5.0
État (0,2)	-8.1	7.1	5.1	-4.0
...				
État (3,2)	0.1	5.0	0.1	5.0
État (3,3)	0.1	5.0	0.1	5.0

Figure 9 : Exemple de Q-table

Chaque case Q[s,a] représente la récompense totale (nommée aussi gain) espérée par l'agent si

- il démarre à l'état s,
- effectue l'action a,
- applique ensuite la politique π.

Ainsi, hors apprentissage, depuis l'état s, la politique d'action mènera à choisir l'action dont le gain espéré est maximal :

$$a = \pi(s) = \max Q[s,:]$$

4.1.2 - Algorithme d'apprentissage

L'algorithme d'apprentissage du Q-learning proposé par Sutton et Barto [2] est alors relativement simple. Il vise à s'approcher de la politique optimale pour maximiser la récompense cumulée. Le livre de Sutton et Barto ajoutera le formalisme mathématique pour les curieux.

Initialiser Q[s,a]
Répéter un nombre N de fois
    initialiser l'état s
    répéter
        choisir action a depuis s en utilisant Q et un peu d'aléatoire
        exécuter l'action a
        observer la récompense r et l'état s'
        mettre à jour Q :
        $Q[s, a] := Q[s, a] + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        s := s'
        jusqu'à ce que s soit l'état terminal ou pour un nombre de boucles max

En regardant le code du fichier FrozenLake_2022.py, on reprend l'ensemble des étapes :

Initialiser Q[s,a]

```
#Création d'un nouvel environnement aléatoire
game = Game(4, 4)
game.print(espace_jeu)
Q = np.zeros([states_n, game.num_actions])
```

La Q-table créée est alors remplie de 0. Il peut être intéressant de remplir la table avec des valeurs aléatoires faibles, pour éviter de privilégier un comportement (celui des actions de faible indice) plutôt qu'un autre, lors des premières étapes d'apprentissage.

Répéter un nombre N de fois

```
#Démarrage de l'apprentissage sur un nombre d'épisodes fixé
for i in range(num_episodes):
```

Un épisode est un ensemble de pas (actions élémentaires) exécutées de l'initialisation à la fin du jeu (carotte ou renard atteint).

La condition d'arrêt de l'apprentissage est ici un nombre d'épisodes. On pourrait aussi choisir d'arrêter l'apprentissage quand il n'y a plus de progrès notable sur la récompense totale obtenue.

Initialiser l'état s

```
#RAZ de l'environnement au début de chaque épisode
actions = []
s = game.reset()
states = [s]
cumul_reward = 0
fin_du_jeu = False
```

La fonction game.reset() remet le lapin à sa position initiale.

Répéter… jusqu'à ce que s soit l'état terminal ou pour un nombre de boucles max

```
#démarrage de l'apprentissage jusqu'à la fin de l'épisode (Renard ou Carotte trouvés)
while True:
```

Sur l'exemple limité Frozen Lake pour laquelle une fin de jeu (le lapin atteint la carotte ou le renard) n'est jamais loin, on peut ne faire se terminer la boucle que sur une fin du jeu :

```
if fin_du_jeu == True:
    break
```

Choisir action a depuis s en utilisant Q et un peu d'aléatoire

```
# on choisit une action aléatoire avec une certaine probabilité, qui décroît petit à petit
#ou on choisit l'action permettant d'espérer la meilleure récompense
if random.random() < facteur_explo*(1. / (i/10 +1)) : #décroissance à régler
    a = random.randint(0,game.num_actions-1)
else :
    a = np.argmax(Q[s,:])
```

Dans l'état s, on remarque le choix de l'action a de manière aléatoire (exploration de nouvelles solutions) ou depuis la Q-table (exploitation des valeurs de la table déjà obtenues).

Le plus souvent, on choisit de diminuer l'exploration au fur et à mesure que s'améliore la Q-table. Le taux d'exploration et le taux de décroissance de l'exploration sont des hyper-paramètres importants pour la vitesse et l'efficacité de l'apprentissage. Trop d'exploration ralentit l'apprentissage et trop peu d'exploration risque de limiter Q à des premières solutions non optimales (minimum local).

Exécuter l'action a

Observer la récompense r et l'état s'

```
#obtention par l'environnement de l'état suivant et de la récompense
s1, reward, fin_du_jeu, _ = game.move(a)
```

La fonction game.move(a) exécute l'action a et renvoie le nouvel état (donné par la fonction de transition), la récompense (reward) et indique si le jeu est terminé (carotte ou renard atteint).

Mettre à jour Q :

$Q[s, a] := Q[s, a] + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
$s := s'$

```
#Mise à jour de la table Q
Q[s, a] = Q[s, a] + lr*(reward + y * np.max(Q[s1,:]) - Q[s, a])
cumul_reward += reward
s = s1
```

L'essentiel du Q-learning est ici, dans la prise en compte de la nouvelle récompense pour améliorer Q. Q[s,a] s'approche de la récompense totale que l'on peut espérer en exécutant a depuis s. On y

trouve donc la récompense obtenue en exécutant a (r) et la récompense totale maximale que l'on pourra espérer dans le nouvel état s' ($\max_{a'}Q(s', a')$).

On note l'introduction de 2 hyper-paramètres : **le taux d'apprentissage** α, nommé lr dans le code (pour learning rate) et **le facteur d'actualisation** □, nommé y dans le code.

La nouvelle valeur de Q[s,a] est une somme de :

- l'ancienne valeur Q[s,a] multipliée par (1-α). On prend ainsi en compte l'ancienne valeur de Q[s,a]. Plus le taux d'apprentissage (α ou lr) est élevé, plus on prend en compte la nouveauté et moins on prend en compte l'existant.
- la récompense (reward) obtenue en exécutant cette action. Elle est multipliée par le taux d'apprentissage (α ou lr), positif et strictement inférieur à 1.
- le maximum de récompense totale espéré dans l'état que l'on vient d'atteindre, multiplié par le facteur d'actualisation (□ ou y) et le taux d'apprentissage (α ou lr). C'est ce qui permet de propager les récompenses importantes vers les états permettant de les atteindre.

Une fois Q actualisée, on copie le nouvel état dans l'état actuel et on recommence.

Le programme permet ainsi d'observer, pas par pas, l'évolution de la Q-table.

Sur la **Erreur ! Source du renvoi introuvable.** est représentée la Q-table en fin d'apprentissage. La carotte étant en (0,0), on voit que l'action HAUT est très valorisée dans la case du dessous (1,0), de même que l'action Gauche dans la case de droite (0,1). On voit également que ces espérances de gain importantes ont « diffusé » vers les cases permettant de les atteindre (2,0), (1,1), (2,1), (2,2)… On peut voir l'inverse avec les actions menant vers le renard (en (1,3) et (2,3) par exemple).

Enfin, il est intéressant de voir que le taux d'exploration assez faible n'a pas permis de tester toutes les actions (l'action D en (0,1) n'a jamais été testée par exemple).

valeurs de la qtable

H 0.00 G 0.00 D 0.00 B 0.00	H -0.85 G 9.78 D 0.00 B 0.00	H 0.00 G 0.00 D 0.00 B 0.00	H 0.00 G 0.00 D 0.00 B 0.00
H 10.00 G 0.00 D 0.00 B 0.00	H 6.18 G -0.85 D -1.57 B -1.57	H -8.50 G -1.71 D -1.69 B -1.94	H 0.00 G 0.00 D 0.00 B 0.00
H 8.90 G 0.00 D 0.00 B 0.00	H -0.98 G 7.81 D -1.67 B -0.85	H -2.51 G 6.73 D -1.71 B -1.00	H -0.98 G -1.69 D -1.69 B -0.98
H 4.45 G 0.00 D 0.00 B 0.00	H -1.69 G -0.85 D -1.57 B -0.85	H -1.91 G -1.00 D -1.69 B -1.69	H -1.57 G -1.71 D -1.69 B -1.69

Figure 10 : Exemple de Q-table en fin d'apprentissage

4.2 – Retour sur les hyperparamètres et autres méthodes tabulaires

La méthode Q-learning étant choisie, les performances de l'apprentissage (rapidité de convergence et caractère optimal de la politique trouvée) sont liées au choix des hyperparamètres. On parle d'hyperparamètres pour les distinguer des paramètres à optimiser que sont les valeurs de la Q-table. On retrouvera ensuite ces hyperparamètres dans le deep Q-learning.

Un taux d'exploration (et un taux de décroissance de celui-ci pas trop élevé) permet d'explorer plus de solutions et d'éviter les minimums locaux. En contrepartie, la convergence sera plus lente.

Un taux d'apprentissage (α ou lr) élevé permet de prendre en compte plus rapidement les nouvelles récompenses obtenues. Par contre, pour des problèmes complexes, il introduit de l'instabilité modifiant trop rapidement les paramètres.

Un taux d'actualisation (ou y) élevé donne plus de poids au gain espéré dans le nouvel état atteint par rapport à la récompense obtenue sur le moment.

Il est possible d'étudier l'influence de chaque hyper-paramètre indépendamment, même si le problème FrozenLake est peu significatif car trop simple.

On insère le code d'apprentissage dans une boucle sur plusieurs valeurs d'un hyperparamètre. A chaque épisode, on stocke la récompense totale obtenue, puis une fois l'apprentissage terminée pour chaque boucle, on trace avec les évolutions de la récompense totale pour chaque valeur de l'hyperparamètres grâce à la bibliothèque matplop.

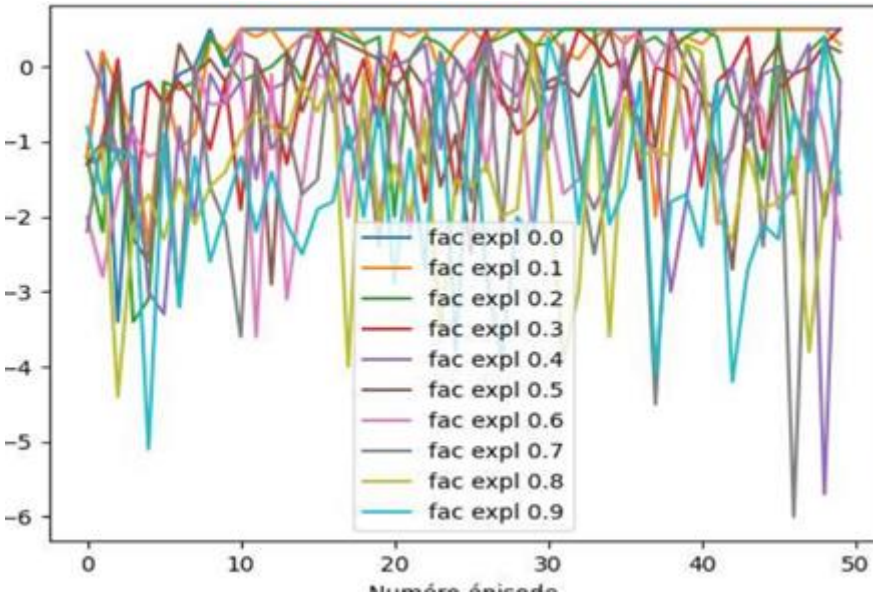

Figure 11 :

Avant d'aborder l'apprentissage par renforcement profond, notons que d'autres méthodes tabulaires existent. SARSA par exemple utilise également une Q-table mais sa mise à jour pendant l'apprentissage est un peu différente du Q-learning.

5 – Apprentissage par renforcement profond

5.1 - Q-apprentissage profond : définition et intérêt

5.1.1 – Du Q-apprentissage traditionnel au Q-apprentissage profond

Pour la plupart des problèmes ayant un environnement possédant un nombre limité d'actions et d'états possibles comme Frozen Lake présenté ci-dessus, le Q-apprentissage « tabulaire » permet d'afficher la Q-fonction sous forme de table. Pour des problèmes possédant un très large nombre d'actions et d'états possibles, il est bien plus coûteux informatiquement, voire impossible, de représenter la Q-fonction dans une table. En effet, dans un environnement comportant n_s états et n_a actions, la table comporte ainsi n_s*n_a cellules ce qui peut, pour des valeurs n_s et/ou n_a élevées, entraîner des coûts computationnels élevés. Ces coûts sont divisés en deux parties, la première étant la quantité de mémoire nécessaire pour sauvegarder et mettre à jour la table qui augmenterait avec le nombre d'états, la deuxième étant le temps nécessaire à l'exploration de chaque état pour créer la Q-table requise.

5.1.2 - Approximation des valeurs de Q par réseau de neurones.

Au lieu de représenter la fonction Q par une table, il est possible de la représenter par une fonction continue. Cette fonction servira d'approximation. N'importe quel type de fonction peut fonctionner : des polynômes, des arbres de décisions… Les réseaux de neurones sont également de bons candidats. Il a ainsi été proposé d'approximer la Q-fonction par le biais de réseaux de neurones comportant des paramètres W (W représente l'ensemble des poids et biais du réseau de neurones), soit $Q(s,a;W) \approx Q^*(s,a)$, Q^* correspondant à la Q-fonction optimale. Un tel réseau de neurones est appelé Q-réseau et prend en entrée l'état et retourne en sortie la valeur de Q de toutes les actions possibles, comme illustré dans la Figure 12.

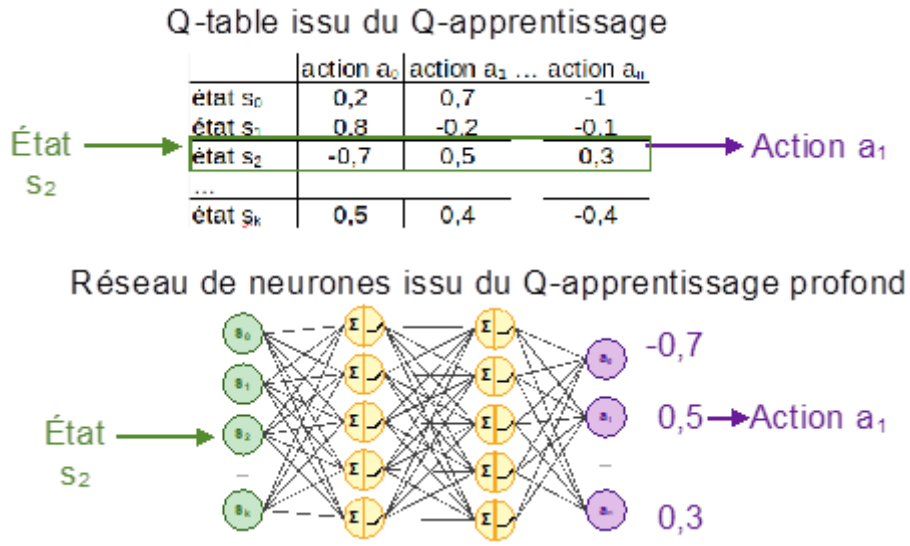

Figure 12 : Q-apprentissage vs. Q-apprentissage profond [5]

5.2 - Défis de l'apprentissage profond par renforcement par rapport à l'apprentissage profond

Nous nous intéressons dans cette section aux différentes étapes de l'apprentissage par renforcement avec l'utilisation d'un réseau de Q-apprentissage profond (deep Q-learning) et aux différences dans l'apprentissage amenées par la nature même de l'apprentissage par renforcement.

En effet, comme présenté dans la ressource « *Introduction à l'apprentissage profond* » [15], l'entraînement supervisé d'un réseau de neurones se fait via un jeu de données déjà étiquetées. Dans l'apprentissage par renforcement, il n'y a pas de jeu de données étiquetées, l'entraînement se fait par les expériences. C'est d'ailleurs un des intérêts de l'apprentissage par renforcement, tout un chacun pouvant travailler sur son problème de type process de Markov, sans besoin d'avoir un énorme jeu de données déjà disponible et donc il n'y a pas d'obligation de se limiter aux problèmes classiques où des jeux de données publics existent.

Cet entraînement par les expériences induit quelques modifications dans l'apprentissage qui seront détaillées ici.

5.2.1 - Apprentissage par renforcement par le biais de réseaux de Q-apprentissage profond

A l'instar d'un réseau de neurones profond, un réseau de neurones de Q-apprentissage profond (cf. Q-réseau) comporte une ou plusieurs entrée(s) et sortie(s), des poids, une fonction de coût, etc. Leurs architectures et fonctionnement sont ainsi similaires.

Dans la partie précédente, le Q-learning utilisait à chaque pas l'équation suivante pour approcher la Q-table de la Q-table optimale :

$$Q[s, a] := Q[s, a] + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Q[s,a] s'approchait ainsi avec le taux d'apprentissage α de la valeur de récompense totale maximale espérée :

- **r** obtenue en exécutant l'action
- **$\gamma \max_{a'} Q(s', a')$**, récompense maximale que l'on pourrait obtenir dans l'état suivant

En apprentissage profond, on cherche aussi à approcher de la récompense maximale espérée **$r + \gamma \max_{a'} Q(s', a';W)$**, d'une manière assez semblable : on va chercher à minimiser, à chaque étape i, une fonction coût L_i égale à :

$$L_i(W_i) = \left[\left(r + \gamma \max_{a'} Q(s', a'; W_i) - Q(s, a; W_i)\right)^2\right]$$

On utilise alors les fonctions d'optimisation présentée dans l'article sur l'apprentissage profond.

A l'instar d'autres réseaux de neurones, l'apprentissage du Q-réseau peut utiliser la rétropropagation du gradient, une méthode consistant à mettre à jour les poids de chaque neurone de la dernière couche vers la première en fonction de l'erreur déterminée par la fonction de coût L_i.

La prise en compte de la modification, à chaque pas, se fait avec un taux d'apprentissage. Si celui-ci est faible, chaque pas apporte une modification mineure aux valeurs du réseau de neurones. On évolue doucement mais on évite des oscillations brutales autour des valeurs intéressantes.

5.2.2 - Le défi du renforcement par apprentissage profond : la nécessité d'un second réseau de neurones

Contrairement à l'apprentissage profond, où la cible (les données étiquetées) est stationnaire menant à un apprentissage stable, nous avons donc une cible (dans **$r + \gamma \max_{a'} Q(s',a';W)$**) non-stationnaire dans le cadre du renforcement par apprentissage profond. Nous sommes donc dans une situation où le Q-réseau calcule à la fois la valeur prédite et la valeur cible, ce qui peut entraîner beaucoup de divergence entre ces deux valeurs. De cette observation est née l'idée d'utiliser deux réseaux de neurones, le Q-réseau et un réseau se chargeant du calcul de la valeur cible, que nous appelons ici réseau cible. Le réseau cible possède la même architecture que le Q-réseau mais avec des paramètres fixes, en pratique, un instantané des paramètres réseau d'il y a quelques itérations au lieu de la dernière itération.

En d'autres termes, un hyperparamètres C est défini, de telle sorte que toutes les C itérations les paramètres du Q-réseau soient copiés et deviennent les nouveaux paramètres du réseau cible comme illustré dans la Figure 13.

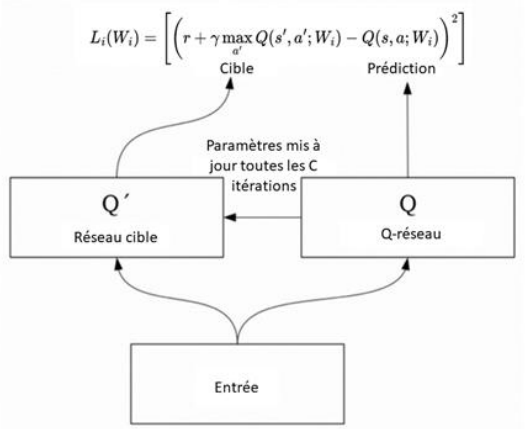

Figure 13: Architecture et fonctionnement de l'approche par deux réseaux de neurones [7]

De ce fait, la fonction cible reste fixe entre chaque C itérations, menant à un apprentissage plus stable.

Dans le cas d'un apprentissage par renforcement par le biais du Q-réseau nommé Q, on retrouve les trois étapes du Q-learning plus la mise à jour du réseau cible Q_{cible} :

1. Dans un état donné, la prochaine action qui sera effectuée est celle ayant la plus grande valeur en sortie du Q-réseau, ou une action aléatoire (exploration qui diminue avec le temps)

2. La fonction de coût est l'erreur moyenne carrée de la différence entre la valeur de Q_{cible} et la valeur de cible.

3. On optimise le réseau Q avec la rétropropagation du gradient

4. Chaque nombre de pas fixé, on recopie Q dans Q_{cible}.

En d'autres termes, en reprenant l'approximation effectuée en 5.1.2 :

$$Q(s,a;W) \approx Q^*(s,a)$$

Elle-même étant issue de l'idée de base du Q-apprentissage, c'est-à-dire l'équation d'optimalité de Bellman comme mise à jour itérative à chaque étape i.

De cette équation, on peut formuler la fonction de coût Li du Q-réseau à chaque étape i (Q_{cible} y est noté Q'):

$$L_i(W_i) = \left[\left(r + \gamma \max_{a'} Q'(s', a'; W_i) - Q(s, a; W_i)\right)^2\right]$$

6 – Exemple pratique de la voiture autonome

6.1 - Présentation de l'environnement

L'exemple Frozen Lake étant trop simple pour mettre en valeur l'apprentissage profond, on propose un second problème, la voiture autonome, avec un modèle physique extrêmement simple permettant des apprentissages en 3 à 15 mn suivant la machine et l'algorithme d'apprentissage.

La voiture dispose de 3 capteurs type télémètres et de 3 actions possibles (à gauche de 3 degrés, tout droit, à droite de 3 degrés). Chaque capteur donne une valeur normalisée entre 1/50 et 1 correspondant à la distance de l'obstacle, en pixels/50.

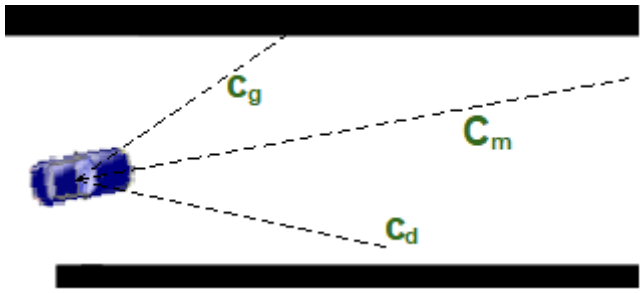

Figure 14 : Description de l'environnement voiture autonome

La fenêtre propose une piste d'entraînement et une piste de validation, pour détecter le sur-apprentissage (cas où la voiture apprend trop bien la piste d'entraînement et ne peut plus généraliser à une autre piste).

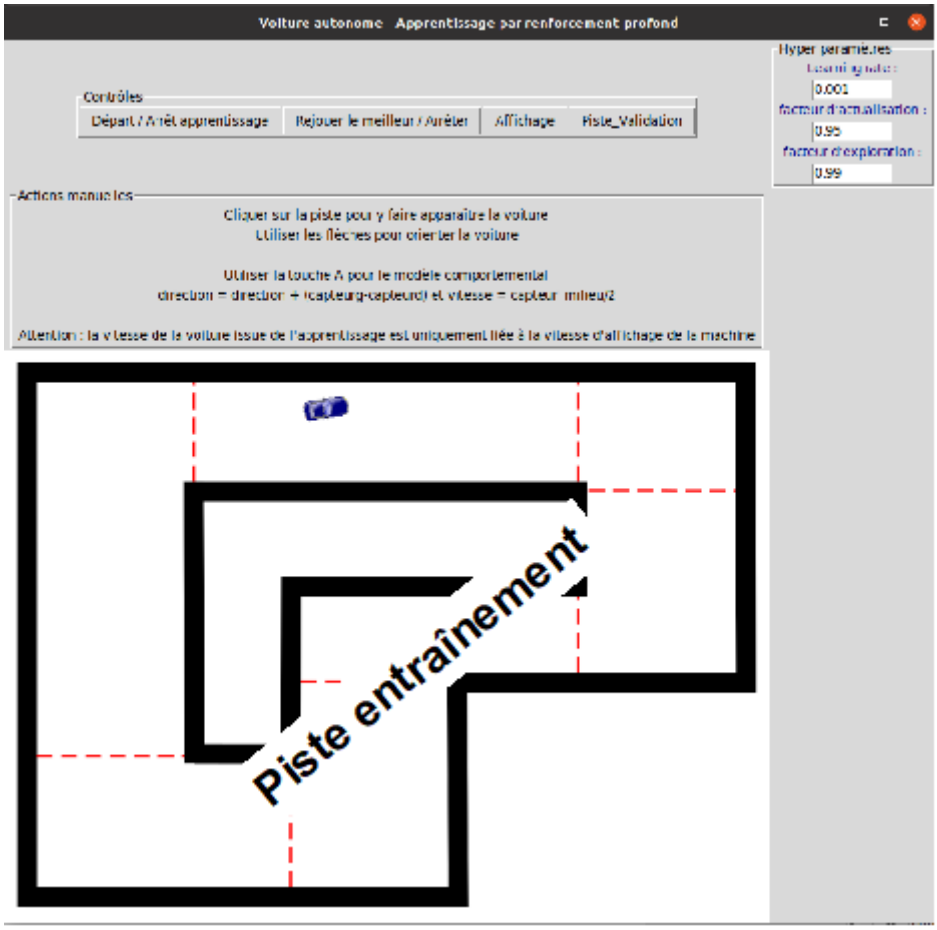

Figure 15 : Fenêtre d'apprentissage avec la piste d'entraînement.
Les pointillés rouge, invisibles de la voiture, sont les balises attribuant +100 de récompense.

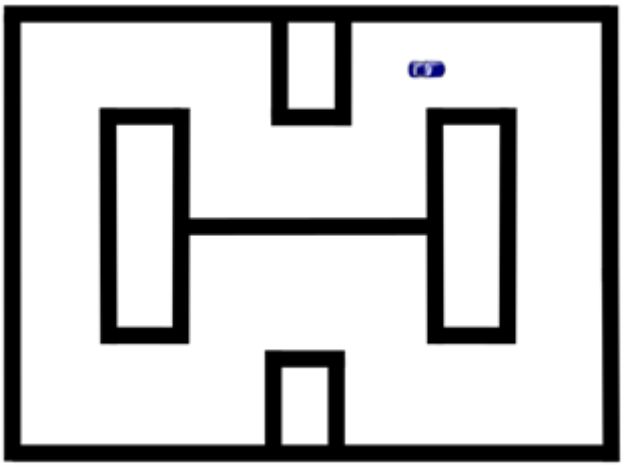

Figure 16: Piste de validation

Voiture_autonome_2022_fenetre_v2.py décrit les fenêtres et boutons, il n'est pas utile de l'ouvrir pour l'exercice. La touche 'A' donne un exemple d'IA symbolique, avec un algorithme comportemental.

Voiture_autonome_2022_voiture_v2.py décrit la classe voiture dans lequel on trouve l'initialisation, les déplacements, les récompenses. La méthode reset de la classe voiture positionne la voiture de manière aléatoire dans une zone de départ pour éviter le sur-apprentissage.

Voiture_autonome_Q_learning_2022_v2 présente un apprentissage par renforcement de type Q_learning, équivalent à celui présenté précédemment, mais avec 50^3 = 125000 états et donc une table de 375000 valeurs {état,action} !

```
#classe décrivant l'agent ayant comme Q-fonction un réseau de neurones pour prendre ses décisions
class #calcul de la sortie du réseau cible. La mémorisation sera désactivée à l'usage
    def forward_target(self, state, remember_for_backprop=True):
    env = None

    #création de l'agent et de son réseau de neurones avec 3 entrées, 2 couches cachées de 24 neurones
    def __init__(self, env):
        self.env = ma_voiture
        self.hidden_size = 24
        self.input_size = 3
        self.output_size = 3
        self.num_hidden_layers = 2
        self.epsilon = 1.0
        self.gamma = 0.95 #taux d'actualisation par défaut

        #CREATION DU RESEAU DE NEURONES PRINCIPAL
        ...
        #CREATION DU RESEAU DE NEURONES CIBLE UTILISE POUR L'OPTIMISATION, à l'identique du principal
        ...
```

On retrouve dans l'initialisation les 2 réseaux identiques Q (main_NN) et Qcible (target_NN)

présente un apprentissage par renforcement profond.

Notons que pour l'affichage de l'image de la voiture avec un angle quelconque, il faut la bibliothèque PIL :

sudo apt install python3-pil python3-pil.imagetk

6.2 - Le réseau de neurones de l'agent de conduite

C'est donc à ce dernier fichier, celui de l'apprentissage profond, que l'on s'intéresse, en soulignant les éléments importants du code.

La Q-fonction de conduite de la voiture associe une valeur aux 3 actions à chaque état du système (les valeurs de ses 3 capteurs). En exploitation, on choisit l'action ayant la plus grande valeur.

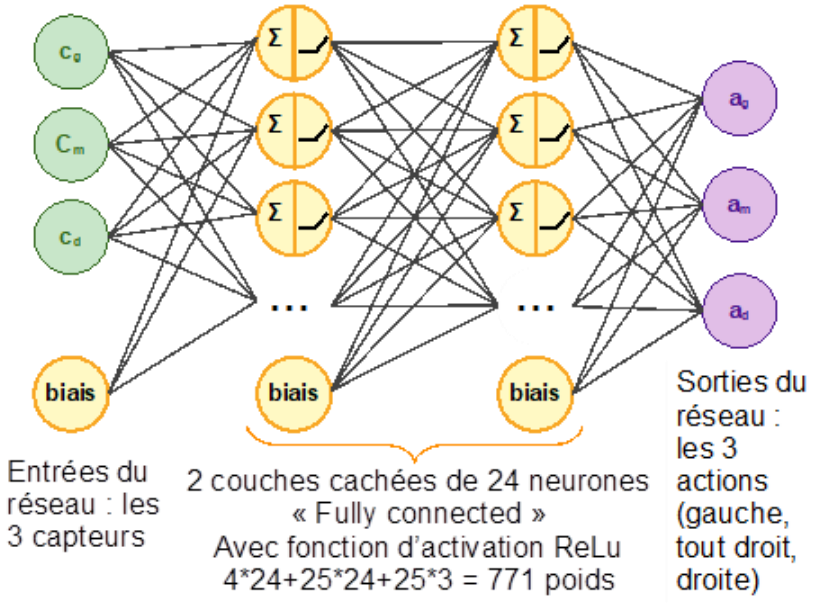

Figure 17 : Réseau de neurones de la Q-fonction « conducteur »

Voiture_autonome_DeepQLearning_2022_v3 contient :

- **La fonction d'activation** et sa dérivée

```
def relu(mat):
    return np.multiply(mat,(mat>0))

def relu_derivative(mat):
    return (mat>0)*1
```

- **La classe NNLayer** pour une couche de neurones, avec les méthodes forward et backward, méthode qui prend en compte la descente de gradient, avec le taux d'apprentissage (learning rate)

```
# classe décrivant une couche de réseau de neurones
class NNLayer:
    # initialisation : nombres d'entrées, nombre de neurones (sorties), fonction d'activation
    def __init__(self, input_size, output_size, activation=None, lr = 0.001):
        self.input_size = input_size
        self.output_size = output_size
        #les poids sont initialisés avec une valeur aléatoire
        self.weights = np.random.uniform(low=-0.5, high=0.5, size=(input_size, output_size))
        self.activation_function = activation
        self.lr = lr
# Calcul des sorties d'une couche à partir des entrées (mode "forward")
def forward(self, inputs, remember_for_backprop=True):
#mise à jour des poids avec le gradient, pondéré par le taux d'apprentissage
def update_weights(self, gradient):
    self.weights = self.weights - self.lr*gradient
#mise à jour de la couche de neurones à partir du gradient venant de la couche suivante
def backward(self, gradient_from_above):
```

- **La classe RLAgent** est celle dont l'instance, nommée model, sera l'agent de notre problème, chargé de prendre les décisions à partir d'un réseau de neurones et de l'optimiser.

```
#classe décrivant l'agent avant comme O-fonction un réseau de neurones pour prendre ses décisions
class #calcul de la sortie du réseau cible. La mémorisation sera désactivée à l'usage
    def forward_target(self, state, remember_for_backprop=True):
    env = None

    #création de l'agent et de son réseau de neurones avec 3 entrées, 2 couches cachées de 24 neurones
    def __init__(self, env):
        self.env = ma_voiture
        self.hidden_size = 24
        self.input_size = 3
        self.output_size = 3
        self.num_hidden_layers = 2
        self.epsilon = 1.0
        self.gamma = 0.95 #taux d'actualisation par défaut

        #CREATION DU RESEAU DE NEURONES PRINCIPAL
        ...
        #CREATION DU RESEAU DE NEURONES CIBLE UTILISE POUR L'OPTIMISATION, à l'identique du principal
        ...
```

On retrouve dans l'initialisation les 2 réseaux identiques Q (main_NN) et Qcible (target_NN)

- La méthode select_action est celle qui permet d'obtenir a depuis s. Le taux d'exploration est réglé par l'attribut epsilon, décroissant au fil des pas.

```
#exploration ou application de la politique pour choisir une action
def select_action(self, state):
```

- La méthode train de la classe RLAgent gère l'apprentissage. Elle utilise les 2 méthodes forward (pour le réseau principal et le réseau cible) pour les calculs des sorties connaissant les entrées, la méthode backward pour l'optimisation des paramètres par rétropopagation de gradient.

```
#entrainement à partir de a, s',s, r : calcul des valeurs d'action données par Q et des
def train(self, done, action, new_state, state, reward):
    action_values = self.forward(state, remember_for_backprop=True)
    next_action_values = self.forward_target(new_state, remember_for_backprop=False)
    experimental_values = np.copy(action_values)
    #application de la formule du deep Q-learning avec -100 si crash ou reward sinon.
    if done:
            experimental_values[action] = -100
    else:
            experimental_values[action] = reward + self.gamma*np.max(next_action_values)

    #Mise à jour des poids par la propagation du gradient vers les couches amont
    self.backward(action_values, experimental_values)
#calcul de la sortie du réseau de neurones en calculant couche après couche,
def forward(self, state, remember_for_backprop=True):
#calcul de la sortie du réseau cible. La mémorisation sera désactivée à l'usage
def forward_target(self, state, remember_for_backprop=True):
#Propagagation du gradient vers les couches amont, mise à jour des poids
def backward(self, calculated_values, target_values):
```

On y retrouve la prise en compte de la récompense et de la récompense maximale de l'état suivant, avec le facteur d'actualisation.

6.3 – l'entraînement

```
# The main program loop
for i_episode in range(NUM_EPISODES):
    #positionnement aléatoire de la voiture au départ
    ma_voiture.reset()
    state = ma_voiture.lecture_capteurs(fenetre.image_piste)
    fenetre.maj_affichage()
    cumul_reward = 0
    index_recopie_target_NN = 0

    # On commence un pas
    while True :
        index_recopie_target_NN +=1
        #on choisit une action en utilisant la politique ou l'exploration (c'est pris
        action = model.select_action(state)
        #on fait un pas et on cumule la récompense
        new_state, reward, done, info = ma_voiture.step(action,fenetre.image_piste)
        cumul_reward += reward

        # On entraine le réseau avec ce pas
        model.train(done, action, new_state, state, reward)
        state = new_state

        #A chaque nombre de pas fixé, on recopie le réseau principal dans le réseau cible
        if (index_recopie_target_NN %20 == 0) : #mise à jour de la NN_target
            for i in range(model.num_hidden_layers+1) :
                model.target_NN[i].weights = np.copy(model.main_NN[i].weights)

        if done:

            #on met à jour la meilleure récompense total et le meilleur réseau
            if cumul_reward > cumul_reward_best :
                for i in range(model.num_hidden_layers+1) :
                    model_best.main_NN[i].weights = np.copy(model.main_NN[i].weights)
                cumul_reward_best = cumul_reward
            break
```

initialisations

Choix d'une action, avec exploration (intégrée par *select_action*)

Exécution d'un pas

Optimisation du réseau

Chaque 20 pas, on met à jour Qcible

Si la récompense est très bonne, on garde une copie du réseau.

L'algorithme de l'entraînement ressemble beaucoup à celui du Q-learning. Validation - Exploitation de l'inférence

Une fois que l'apprentissage paraît satisfaisant, arrêter l'apprentissage puis lancer via le bouton Rejouer l'exploitation du meilleur réseau.

```
#gestion du rejeu de l'inférence
if fenetre.rejouer :
    fenetre.rejouer= False

    state ,_,_,_ = ma_voiture.step(0,fenetre.image_piste)
    while(True): #tant qu'il n'y a pas d'accident
        action = model_best.select_action(state)
        state ,_,_,_ = ma_voiture.step(action,fenetre.image_piste)
        fenetre.maj_affichage()
        if(state[1]<0.02) : #en cas d'accident on arrête la simulation
            break
        if fenetre.start == True :
            fenetre.start = False
            break
        if fenetre.rejouer :
            fenetre.rejouer= False
            break
```

Il est possible de placer avec la souris la voiture n'importe où sur la piste d'entraînement comme sur la piste de validation.

Les hyperparamètres choisis pour le Q-learning notamment conduisent à une table Q qui permet de tourner sur la piste d'entraînement mais pas sur la piste de validation (un demi-tour maximum).

6.4 – Exercices envisagés

L'environnement maîtrisé, il est possible d'affiner le système en augmentant :

- le nombre d'actions possibles (qui restent discrètes),
- le nombre de télémètres (on peut se rapprocher d'un Lidar à 200 faisceaux),
- la résolution de ces télémètres…

On pourrait aussi envisager d'améliorer la physique de la voiture (adhérence, inertie, rotation type « bicyclette », …), avec le risque d'augmenter le temps de calcul. Pour plus de réalisme, on peut consulter l'article « *Apprentissage par renforcement de la conduite d'un véhicule* » [14].

On peut également imaginer faire courir 2 voitures sur la même piste. On peut faire de nouvelles pistes (fichier .png noir & blanc)

Un autre aspect intéressant, notamment dans une classe composée de nombreux étudiants-expérimentateurs, est d'étudier l'influence des hyper-paramètres en traçant l'évolution de la récompense cumulée pour différentes variations de chaque hyper-paramètre :

- taux d'apprentissage
- facteur d'actualisation
- facteur d'exploration,
- nombre de pas rythmant la mise à jour du réseau cible,
- vitesse à laquelle diminue le taux d'exploration,
- nombre de neurones par couche et nombre de couches,
- les récompenses utilisées.

Notons qu'il est simple de modifier la méthode *reset* de la classe voiture pour supprimer le positionnement aléatoire au démarrage et ainsi mettre en évidence le sur-apprentissage (la voiture ne fonctionne qu'en partant de ce point).

7 – Influence de l'environnement sur l'apprentissage par renforcement profond

Maintenant que nous avons clairement défini l'architecture et le fonctionnement de nos deux réseaux de neurones, il est d'intérêt de se pencher sur l'impact de l'environnement sur l'apprentissage de notre agent. En effet, nous avons ici particulièrement présenté deux problèmes, Frozen Lake et la voiture autonome. Leurs environnements sont très différents ainsi que l'apprentissage dans chacun de ces cas. Le principal aspect que nous voulons mettre en avant est la différence entre ces environnements et la similarité de leurs états interdépendants. Ce sujet est particulièrement important car directement corrélé à l'évolution de la complexité de l'environnement des problèmes étudiés et donc principalement présent dans les problèmes d'apprentissage par renforcement profond.

7.1 - Similarité des états interdépendants

Pour illustrer ce que nous appelons ici la similarité des états interdépendants, nous utilisons de nouveau l'exemple de la voiture autonome. L'objectif est de lui apprendre à conduire sur un circuit. Imaginons que le début du circuit soit une longue ligne droite. Dans ce cas, l'agent n'aura rien appris pour la gestion de courbes. Contrairement, un circuit totalement circulaire n'apprendra pas à notre agent à se déplacer en ligne droite. De ce fait, si les agents cités au-dessus sont déposés sur le circuit de l'autre, ils ne pourraient aucunement le parcourir. Pour contourner cet effet de

l'apprentissage par renforcement dit « en ligne » (l'agent peut interagir comme bon lui semble avec l'environnement), nous nous tournons vers l'apprentissage par renforcement dit « hors ligne ». Ces deux types d'apprentissage sont illustrés dans la Figure 18.

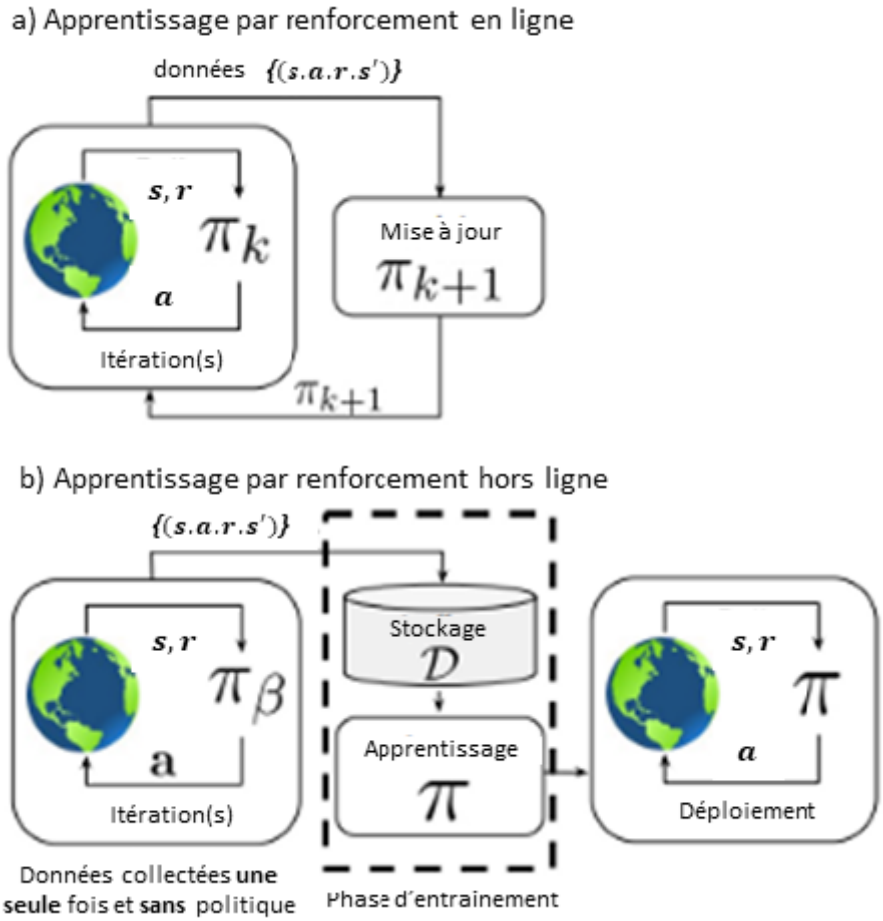

Figure 18 : Principes de fonctionnement : a) l'apprentissage par renforcement en ligne, b) l'apprentissage par renforcement hors ligne [8], π_θ est juste un embryon de politique.

7.2 - Apprentissage par renforcement hors ligne

L'apprentissage par renforcement purement hors ligne empêche l'agent d'interagir avec l'environnement. C'est utile par exemple pour travailler avec des données réelles de véhicule issues d'essais avec un conducteur (qui applique sa propre politique π_β). Le conducteur essaiera alors de passer dans le plus d'états possibles.

L'agent apprend donc maintenant depuis un lot d'expériences, qui est aléatoirement échantillonné, duquel l'agent sélectionne un échantillon uniformément distribué et effectue son apprentissage grâce à lui.

L'échantillonnage aléatoire des expériences permet ainsi de réduire largement le biais introduit par la possible nature séquentielle d'un environnement, représenté par l'exemple de la ligne droite.

7. 3 - Mémorisation pour l'apprentissage par renforcement hybride

Une autre méthode utilisée pour l'apprentissage par renforcement, nommée experience replay, est hybride entre apprentissage en ligne et hors-ligne.

Comme nous l'avons vu, il n'est pas souhaitable d'optimiser le réseau Q en utilisant pour cible des données issues de ce réseau Q.

Une alternative à l'utilisation d'un réseau cible est l'utilisation d'expériences mémorisées pour l'optimisation.

A chaque essai, on enregistre l'expérience (s, a, r, s') et on fait l'optimisation sur des données pris aléatoirement dans la base de données d'expériences.

Un exemple commenté d'apprentissage par renforcement profond avec experience replay est présenté dans le fichier : Voiture_autonome_DeepQLearning_2022_experience_replay.py

8 – Conclusion

Dans cette ressource ont été mis en avant les fondements de l'apprentissage par renforcement et de l'apprentissage par renforcement profond. Des exemples généralistes et largement étudiés ont été utilisés pour illustrer les propos tenus. Cependant, nous ne pouvons que souligner la non exhaustivité de la liste des applications possibles montrées ici. De la même façon, les algorithmes mis en avant dans cette ressource sont nécessaires à la compréhension des premiers pas à la fois de l'apprentissage par renforcement tabulaire et profond.

De nombreux autres algorithmes et approches ont ainsi émergés, outrepassant certaines limites inhérentes aux approches présentées. Pour n'en citer que les plus fameuses d'entre elles, le Deep Q-Network (DQN) est l'évolution direct du Q-Apprentissage Profond étudié, le Proximal Policy Optimization (PPO), qui se décline en 2 variantes majeures qui toutes les deux offrent en règle générale de meilleures performances et une meilleure convergence. Cette méthode est cependant peu robuste car sensible aux changements. Dans un autre registre, le Soft-Actor Critic (SAC) est très efficace pour les techniques d'optimisation basées sur l'énergie en raison de la régularisation du facteur entropique. En d'autres termes, SAC utilise la régularisation entropique, la politique étant entraînée à maximiser un compromis entre la valeur de Q prédite et l'entropie. Ces trois approches correspondent aux méthodes principalement utilisées dans l'état de l'art actuel et leur intérêt pratique étant évident, il est conseillé de s'intéresser à leur fonctionnement, par exemple en s'aidant des ressources disponibles en [9].

Ecrire les optimisations des réseaux de neurones a permis de bien comprendre leur fonctionnement. Pour le travail sur des problèmes importants, avec les algorithmes DQN, PPO ou SAC, StableBaseLine est un outil très populaire. Il est présenté dans un autre article, avec l'environnement Gym. Un troisième article utilise Gym, StableBaseLine et le simulateur AirSim pour un simulateur de voitures autonomes réaliste.

Enfin, on a considéré ici l'observation o de l'environnement par l'agent comme étant son état (s = o). Cependant, dans certains travaux, les deux sont distingués. En effet, dans certains types d'apprentissage par renforcement plus complexes, l'état est « partiellement observable ». On parle de processus de Markov partiellement observables (POMDP). Ainsi, on essaiera de déduire l'état à travers les observations générées. C'est le cas notamment à travers les travaux sur l'Inférence Active mené par Karl Friston [8] (qui consiste à caractériser les comportements des individus via leurs émotions et leur perceptions).

Une fois bien maîtrisé ces notions, il est maintenant possible d'adapter son propre problème à un environnement markovien pour permettre l'apprentissage par renforcement. Pour les systèmes réels complexes, l'œil de l'expert permettra de choisir les bonnes approximations. Une activité intéressante est alors le passage de l'inférence du réseau de neurones obtenue en simulation sur le

système réel. Des méthodes nommées Sim2Real existent pour rendre plus robuste ce passage. Un exemple de voitures autonomes est accessible sur le site culture Science de l'ingénieur : https://eduscol.education.fr/sti/si-ens-paris-saclay/actualites/course-de-voitures-autonomes-2022-resultats

Références :

[1] https://www.deepmind.com/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules

[2] R. S. Sutton, A. G. Barto, Reinforcement Learning An Introduction : Second Edition, MIT Press, 2014 https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

[2] La documentation de Stablebaselines, https://stable-baselines.readthedocs.io/en/master/

[3] L. Da Costa, T. Parr, N. Sajid, S. Veselic, V. Neacsu, K. Friston, « Active inference on discrete state-spaces: A synthesis », Journal of Mathematical Physchology, 99, 2020.

[4] La documentation de l'environnement Gym, https://www.gymlibrary.ml/

[5] https://www.mlq.ai/deep-reinforcement-learning-q-learning/

[6] https://www.tensorflow.org/agents/tutorials/0_intro_rl

[7] https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[8] https://ichi.pro/fr/decisions-a-partir-des-donnees-comment-l-apprentissage-par-renforcement-hors-ligne-changera-la-facon-dont-nous-utilison-40516786874800

[9] https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1

[10] Introduction à l'apprentissage par renforcement, A. Juton, V. Noël, R. Lali, (dépôt des fichiers source), https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/introduction-a-lapprentissage-par-renforcement

[11] Dossier Intelligence Artificielle, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/dossier-intelligence-artificielle

[12] Stabilisation d'un pendule inversé à l'aide d'un apprentissage par renforcement, G. Chérot, M. Gallois, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/stabilisation-dun-pendule-inverse-alaide-dun-apprentissage-par-renforcement

[13] Introduction aux bibliothèques Gym et Stable Baselines pour l'apprentissage par renforcement, G. Chérot, A. Godinot https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/introduction-aux-bibliotheques-gym-et-stablebaselines-pour-lapprentissage-par-renforcement

[14] Apprentissage par renforcement de la conduite d'un véhicule sur AirSim, L. De Matteïs, S. Radosavvljevic, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/apprentissage-par-renforcement-dela-conduite-dun-vehicule-sur-airsim

[15] Introduction à l'apprentissage profond, S. Janny, L. De Matteïs, W. Shu-Quartier, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/introduction-lapprentissage-profond