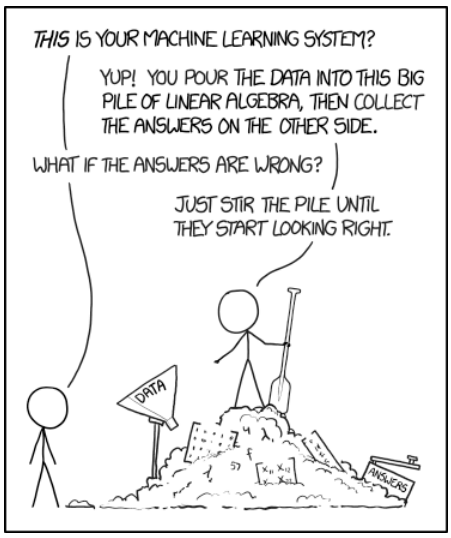
Rapport de projet départemental

Apprentissage par renforcement et contrôle de la marche du robot

FICM 2A – 2022/2023



Machine learning
https://xkcd.com/1838/

Étudiant en charge du projet : Lucas Bertinchamp

Tuteurs du projet : Patrick Hénaff et Loïck Briot

Table des matières

Introduction 3
Partie I - Mise en place des simulateurs 4
I.1 - Simulateur 3D pybullet 4
I.2 - Caméra et obstacles 5
I.3 - Difficultés et passage à la 2D 6
I.4 - Simulateur en deux dimensions avec Pygame 6
I.5 - Gestion de la physique avec Pymunk 7
Partie II - L'apprentissage par renforcement, principe et utilisation 9
II.1 - Origine et principe de l'apprentissage par renforcement 9
II.2 - Vocabulaire de l'apprentissage par renforcement 10
II.3 - Environnements d'apprentissage 11
II.4 – Choix de l'algorithme d'apprentissage 14
II.4.1 – Deep Q-Network 15
II.4.2 – Proximal Policy Optimization 16
II.5 - Optimisation des hyperparamètres 16
II.6 - Importance et choix de la reward 18
Partie III - Étude des résultats 21
III.1 - Calculs effectués à distance 21
III.2 - Affichage des résultats avec tensorboard 21
III.3 – Résultats d'apprentissage 22
III.3.1 : Modèle éparse 23
III.3.2 : Modèle dense 24
Conclusion 26
Bibliographie 27

Introduction

La deuxième année à l'École des Mines m'a donnée la possibilité de choisir un projet scientifique parmi de nombreux sujets proposés en tant que projet départemental. C'est dans un large éventail de thématiques que j'ai décidé de choisir le projet intitulé « Apprentissage par renforcement et contrôle de la marche du robot » car celui-ci a su éveiller ma curiosité.

En effet, l'intelligence artificielle et l'apprentissage sont des sujets qui m'intéressent depuis plusieurs années et ce projet me donne l'occasion de les mettre en pratique. De plus, le cours « Introduction à l'apprentissage automatique » du tronc commun de deuxième année a donné les bases théoriques de l'apprentissage mais en laissant sous silence l'apprentissage par renforcement. C'était donc le moment adéquat pour développer mes connaissances sur une des branches principales du machine learning qui n'avait pas encore été abordée en cours. Enfin, le côté robotique de ce projet me permettrait de rendre plus concret les objets crées en simulateur sur ordinateur en permettant à des robots de se déplacer.

L'objectif du projet est donc le suivant : permettre à un robot de se déplacer dans son environnement afin d'atteindre une cible, tout en évitant les obstacles. Ce robot doit être capable de se déplacer en prenant des décisions par lui-même ; ainsi on utilisera l'apprentissage par renforcement afin de l'entrainer à réussir sa tâche.

L'énoncé précédent paraît simple, mais cache un projet extrêmement vaste qu'il faut décortiquer étape par étape. Entre simulations, apprentissage par renforcement, robotique … les concepts à découvrir sont nombreux, surtout quand on démarre presque de 0 comme cela fut mon cas en début d'année. A raison d'une demi-journée par semaine il semble difficile de terminer complétement ce projet, mais j'ai fait tout mon possible pour aller le plus loin afin que la tâche restante à accomplir puisse être prise en charge facilement si quelqu'un décide de reprendre mon travail l'an prochain.

Ainsi, la suite de ce rapport présente l'ensemble du travail que j'ai pu accomplir cette année. Il développe les outils mis en place, les concepts abordés, les choix faits et les difficultés rencontrées. J'ai pris la décision d'écrire la suite de ce document dans un ordre logique plutôt que chronologique afin que quiconque puisse suivre la lecture sans difficulté et comprendre au mieux les notions abordées. (L'ordre chronologique n'étant pas si éloigné de ce qui va suivre)

C'est pourquoi je propose de commencer par présenter les simulateurs qui ont été mis en place. En effet, avant de parler apprentissage ou robotique il est important de modéliser notre problème et d'essayer de le résoudre sur ordinateur et cela passe par la conception de simulateurs. Ensuite, je détaillerai le fonctionnement de l'apprentissage par renforcement en commençant par le principe de cette méthode et les algorithmes utilisés jusqu'à l'optimisation des hyperparamètres du modèle. Enfin, je présenterai les résultats que j'ai pu obtenir, comment les interpréter et quelles pistes aborder pour aller plus loin.

Partie I - Mise en place des simulateurs

I.1 - Simulateur 3D pybullet

Avant de travailler sur un robot réel, il est nécessaire de mettre en place un simulateur dans lequel notre robot simulé pour évoluer en suivant le code qu'on a préalablement développé. Cela permet de tester notre travail et d'éviter les mauvaises surprises lors des tests en condition réelle. Plusieurs choix de simulateurs s'offrent alors à nous selon la nature du projet.

Dans mon cas, la volonté était à terme de contrôler le robot Unitree Go1. C'est un robot quadrupède capable de nombreuses actions pour évoluer dans son environnement. Il fallait donc mettre en place un simulateur 3D pour pouvoir tester le comportement du robot. Après plusieurs jours de difficultés à installer Gazebo et ROS (Robot Operating Systems), nous avons penché pour Pybullet, une librairie python permettant de simuler des environnements 3D et d'interagir avec eux. Comme base, nous nous sommes inspirés du github suivant : https://github.com/OpenQuadruped/spot_mini_mini, qui mettait en place le « spot mini », un robot similaire à celui d'Unitree, et permettait de le contrôler à l'aide d'une interface permettant de faire varier ses différents paramètres (figure I.1-1). Les plus importants d'entre eux étant « Step Length » qui permettait de faire accélérer le robot, ainsi que « Yaw Rate » permettant d'incliner le robot sur son coté et donc lui permettre de tourner.

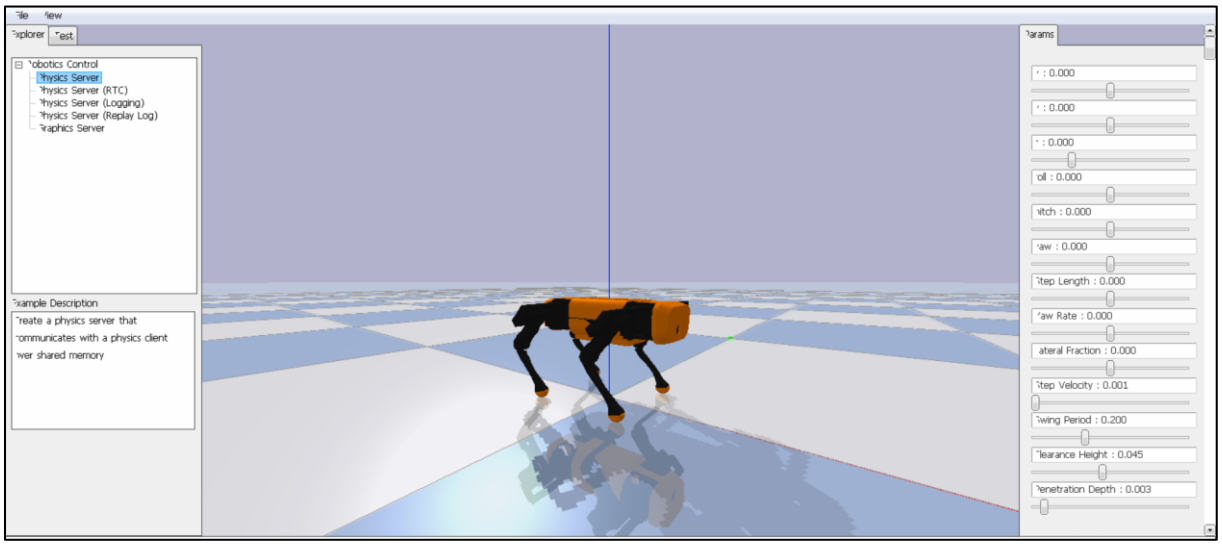


Figure I.1-1 : Interface de pybullet avec le robot spot mini

La première étape fut donc de prendre en main le code de cette simulation. D'ailleurs, l'intégralité du code de mon projet peut être retrouvé sur le github de Mines Nancy à l'adresse suivante : https://github.com/mines-nancy/projet_lucas_bertinchamp.

Afin de pouvoir déplacer le robot dans son environnement, j'ai abandonné l'interface initialement développée pour mettre en place un contrôle par les flèches directionnelles du clavier. Cela permet de contrôler plus facilement le robot en lui indiquant une action à effectuer (fais un pas en avant, en arrière, tourne à gauche ou à droite) plutôt que de lui communiquer une information de vitesse et d'orientation. De plus, il faut gérer les collisions du robot avec les objets du simulateur en lui permettant de les détecter. Cela est important dans la mesure où l'algorithme d'apprentissage par renforcement va avoir besoin de connaître les interactions entre le robot et les obstacles sur son chemin. De tels outils permettant de connaitre les distances entre les objets se trouvent dans la bibliothèque Pybullet.

I.2 - Caméra et obstacles

Toujours dans une optique de donner du lien entre le robot et son environnement simulé, il était nécessaire d'implémenter une caméra pour que le robot obtienne des informations de distances. Il existe plusieurs types de caméra, la plus connue étant la caméra RGB qui renvoie une image en couleurs. Ici, nous utilisons une caméra de profondeur qui, à chaque pixel de l'image renvoyée, communique une information de distance à l'objet le plus proche.

Le choix d'ajouter une caméra de profondeur est liée au fait que Unitree Go1 possède une caméra sur sa face avant dont nous pourrons récupérer les informations. De plus, les données de cette caméra sont sous la forme d'un vecteur unidimensionnel (une valeur par pixel étant une distance) et qui peut donc facilement être interprété par le réseau de neurones de l'algorithme d'apprentissage. Pour des robots non équipés de caméra, on peut aussi envisager d'utiliser un LIDAR qui renvoie des distances autour du robot à l'aide de lasers envoyés dans toutes les directions. Pybullet permet de générer des caméras qui renvoient les images à intervalle de temps régulier à l'aide de matrice de vue et de projections.

Ensuite, il a fallu implémenter les obstacles. L'objectif du projet étant de permettre l'évitement d'obstacles il était important de mettre rapidement en place de tels objets. Dans un simulateur 3D, chaque objet est représenté par un fichier URDF qui comprend toutes les caractéristiques de ce dernier. Ainsi, on peut créer des fichiers URDF pour chaque type d'obstacles que l'on veut mettre en place. J'ai décidé de créer différents types de murs : un mur droit, un mur en forme de « L » pour les coins et un mur en en forme de « + ». De plus, j'ai pu mettre en place un petit outil qui permet de dessiner dans un fichier texte les obstacles que l'on veut créer et qui charge automatiquement les fichiers correspondant dans la scène. Un exemple est illustré dans la figure I.2-1.

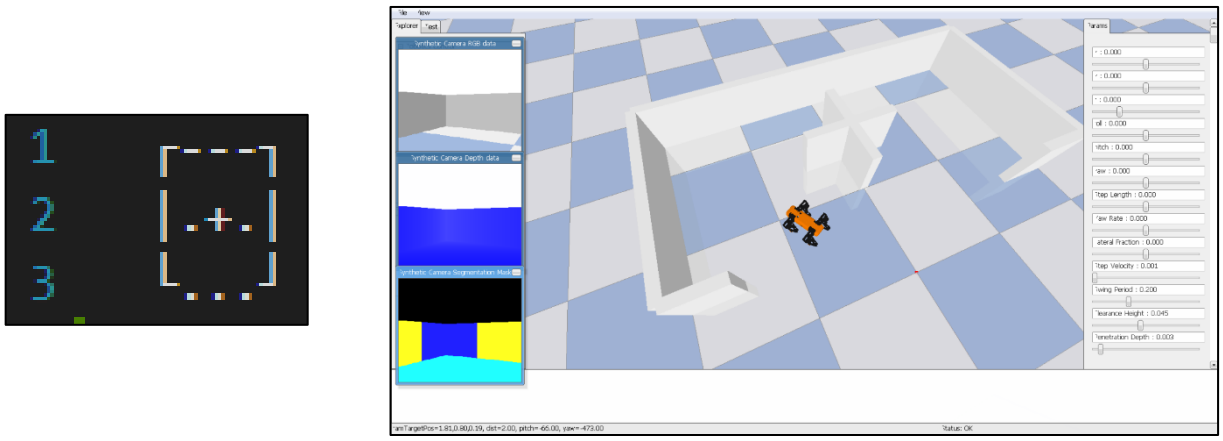


Figure I.2-1 : Création d'obstacles sous Pybullet et images rendues par caméra

La figure I.2-1 montre également les images générées par la caméra du robot placée juste devant lui. La caméra que nous utiliserons est la deuxième, la caméra de profondeur.

I.3 - Difficultés et passage à la 2D

Maintenant que les différents outils sont en place, il a fallu mettre en place les environnements d'apprentissage par renforcement. Je reviendrai en détail sur ce point dans la deuxième partie de ce document. À la suite de plusieurs semaines de travail, le premier apprentissage par renforcement à pu être lancé, l'objectif étant simple : apprendre au robot à avancer tout droit d'un mètre. C'est une tâche qui parait simple étant donné de la facilité de l'action décrite et de la distance à parcourir. Pourtant, il a fallu un peu plus d'une heure à notre robot pour apprendre à faire cette tâche.

Ce premier test a soulevé plusieurs problèmes. De toute évidence**, l'apprentissage par renforcement est un processus lent,** apprendre une tâche même basique semble complexe et d'autant plus quand le champ d'action offert par la simulation est vaste. De plus, le simulateur était extrêmement lent ; simuler à chaque instant un robot dans un environnement 3D, faire le rendu de ses caméras ou encore gérer les collisions avec les obstacles sont des processus très couteux qui ralentissent d'autant plus l'apprentissage. Malgré le temps passé sur pybullet, il ne semblait pas possible de continuer sur ce simulateur sans disposer d'ordinateurs suffisamment puissant ou de beaucoup de temps.

Ainsi, j'avais besoin d'une autre façon d'aborder notre problème. L'environnement 3D étant couteux en termes de calcul, alors passer dans un environnement 2D ne serait-il pas judicieux ? En effet, notre robot simulé en 3D n'évolue pas selon l'axe Z et reste donc dans un plan parallèle au sol. En considérant ce point, si nous arrivons à créer un système en deux dimensions suffisamment « proche » de celui en trois dimensions, alors l'apprentissage effectué en 2D pourrait être également utilisé en 3D. Par « proche » on entend que le problème traité dans les deux cas soient identiques, que le robot utilisé en 2D et en 3D possède le même type de déplacement, que la détection des obstacles soient traités d'une façon similaire etc… Si passer en 2D permet d'apprendre au robot à résoudre une tâche plus complexe beaucoup plus rapidement et que les résultats peuvent être utilisés en 3D, alors c'est une piste à suivre. Ainsi, dans toute la suite de ce document, nous nous intéresserons au simulateur en deux dimensions pour mettre en place un robot « intelligent » capable d'éviter les obstacles.

I.4 - Simulateur en deux dimensions avec Pygame

Pour mettre en place un tel simulateur à partir de zéro, mon choix s'est tourné vers **Pygame**, qui comme son nom l'indique, est réputé dans la création de petits jeux en Python. Il permet de gérer tout le rendu graphique ainsi que les événements entre le joueur et le jeu (clic de souris, appuyer sur une touche du clavier etc …). Cette librairie fonctionne en mettant en place une boucle du jeu dans laquelle toutes les actions s'effectuent.

Pour mettre en place un environnement en deux dimensions similaire à celui en trois dimensions nous avons besoin de :

- **Un robot** : qui sera représenté par un cercle dans le simulateur en deux dimensions. Celui-ci doit disposer d'actions similaire à celui de pybullet. Ainsi, il possédera deux paramètres étant sa vitesse et son angle de rotation. A chaque instant il doit être capable d'avancer dans la direction décrite par son angle. En définissant un tel angle, cela

permettra au robot de tracer des trajectoires circulaires comme pourrait le faire un robot réel.

- **Une cible** : qui sera représenté par un point rouge sur l'écran. Il suffira de décrire la position sur les axes X et Y pour afficher ce point.
- **Des obstacles** : que l'on pourra afficher sous la forme de rectangles gris de dimensions variables à l'écran.
- **Un système de traitement des distances** : nous mettrons en place ce qui peut s'apparenter à un LIDAR. Ce dispositif tirera un nombre n de lasers autour du robot avec un écart réguliers entre eux et renvoyant la distance à l'obstacle le plus proche.

Commençons par les deux premiers points qui peuvent facilement être mis en place avec pygame. En effet, pour ces deux objets, il suffit de créer un Sprite que l'on demande à afficher à chaque instant dans la boucle pygame. L'écran doit être réinitialisé à chaque tour de boucle pour éviter d'avoir plusieurs fois le même Sprite affiché.

Concernant les déplacements, il suffit de demander à pygame de récupérer les évènements lorsque l'utilisateur appuie sur les touches de son clavier. Une pour augmenter la vitesse, une pour la diminuer, une pour tourner à gauche et une pour tourner à droite.

I.5 - Gestion de la physique avec Pymunk

La gestion de la physique du simulateur se fera avec **Pymunk**, une bibliothèque Python permettant d'implémenter **des corps rigides** (rigid body) et de contrôler leurs interactions entre eux. Pour se faire, on utilise l'objet Space de pymunk qui permet de contenir l'ensemble des objets dotés de physique. Pour chacun de ces objets, on peut leur donner des propriétés physiques (vitesse, friction, élasticité, effet de la gravité etc…) qui seront prises en compte dans le calcul des interactions entre chacun d'eux. Ainsi, pour chaque itération de la boucle pygame, on pourra enclencher la fonction Step de cet espace d'objet pour faire avancer d'un pas l'ensemble des objets de notre scène.

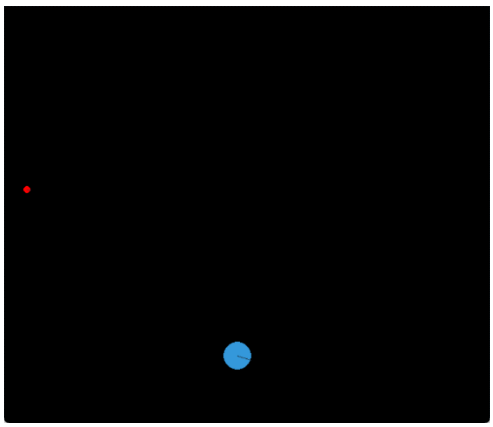
Grâce à cette bibliothèque, il est possible de s'occuper des 2 derniers points de notre liste. Notre espace d'objets physique sera constitué du robot, de la cible ainsi que des obstacles présents dans la scène. On remplace notre classique Sprite pygame du robot, par un **Circle** et un **Body** pymunk qu'on initialise à une vitesse nulle. On procède de même pour la cible. Même si cette dernière reste statique, on l'a munie de physique pour éviter qu'elle se trouve dans un obstacle lors de l'initialisation de l'environnement.

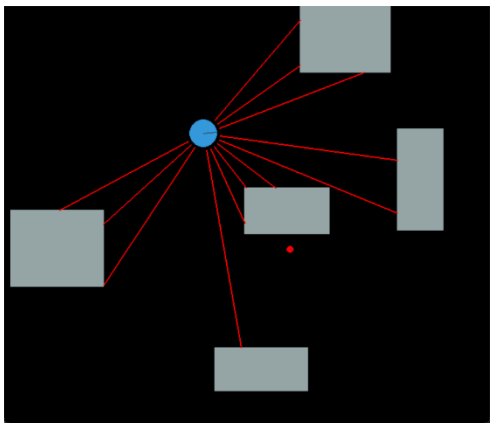
Pour les obstacles, on crée **des Body statiques** additionné à un objet **Poly** pour leur donner la forme voulue. De cette façon, on crée aléatoirement des obstacles dont la hauteur et la largeur sont compris entre 25 et 75 pixels. Le nombre d'obstacles par scène sera généralement compris entre 0 et 5. Ensuite, on implémente une fonction qui se déclenchera lorsque deux objets seront détectés comme étant en collision. Pour cela, on crée au préalable des groupes d'objets : les obstacles ensemble, le robot d'autre part. Ainsi, on évite de déclencher cette fonction si deux obstacles sont en collision au moment de leur génération (ce qu'on peut interpréter comme un obstacle d'une forme plus complexe ou deux obstacles mis bout à bout).

Finalement, il faut implémenter le système de détection de distance. On utilise la **fonction segment_query_first** qui agit sur l'espace d'objet physique. En lui donnant un point de départ et un point final, il trace un segment et détermine le premier point de contact entre ce segment et un objet physique. Dans notre cas, on tracera des segments de taille 500 pixels en considérant qu'au-delà de cette distance, il n'est pas nécessaire de considérer l'obstacle. Autrement, on récupèrera les distances

entre le robot et les points de contacts déterminés par ces jets de segments. Le nombre de segments tirés peut être paramétré, chacun d'entre eux sera espacé de manière régulière.

Désormais, tous les éléments sont en place pour notre simulateur 2D. Les figures I.5-1 et I.5-2 présentent son affichage dans le cas où aucun obstacle est à portée, et dans le cas où un obstacle est détecté par les lasers. Le robot est représenté par le cercle bleu, la cible par le rouge. Les obstacles sont les rectangles gris et les lasers sont représenté par un trait rouge quand un obstacle est détecté. Nous pouvons désormais passer à la mise en place de l'environnement d'apprentissage.





Figures I.5-1 et I.5-2 : Affichage du simulateur 2D sans et avec obstacles

Partie II - L'apprentissage par renforcement, principe et utilisation

II.1 - Origine et principe de l'apprentissage par renforcement

L'apprentissage automatique (machine learning) est une approche statistique qui permet de doter aux ordinateurs d'une capacité « d'apprentissage » à l'aide de données. Ainsi, pour un problème donné, une machine sera capable de le résoudre après avoir appris en traitant un certain nombre de données que l'on lui aura donné au préalable (on parle alors de phase d'apprentissage). La nature de l'apprentissage dépend du format des données. Si ces données sont non étiquetées, on parle **d'apprentissage non-supervisé** : par exemple trier des images afin d'en créer des groupes distincts. (on parle souvent de problème de classification ou de partitionnement) Si au contraire les données sont étiquetées, on parle **d'apprentissage supervisé** : par exemple la reconnaissance d'un chiffre sur une image (chaque image est étiquetée par le chiffre qu'elle représente et que l'ordinateur doit reconnaître).

L'apprentissage par renforcement se présente comme la 3ème branche principale du machine learning. Ici, c'est la machine qui va créer ses propres données en évoluant dans son environnement. En effet, la machine aura accès au cours de son fonctionnement, à son état actuel. À partir de cet état, la machine détermine une action à effectuer pour se rapprocher de son objectif (action souvent déterminée par **un réseau de neurones**). Lors de la phase d'apprentissage, pour savoir si un état est meilleur qu'un autre, on utilise un système de **récompense** qui est un nombre que l'on donne à la machine. Plus cette valeur est élevée, plus elle comprend que son état actuel est meilleur qu'un autre. Ainsi, l'objectif d'une machine entrainée par renforcement est de trouver une suite d'action à effectuer afin de maximiser la récompense qu'il peut recevoir. Les différents **algorithmes d'apprentissage** s'occupent d'ajuster les paramètres du réseau de neurones en fonction des récompenses reçues afin que la machine devienne de plus en plus performante à réaliser la tâche en question.

L'apprentissage par renforcement est une méthode d'apprentissage qui a pris de l'ampleur assez récemment. Elle est utilisée dans beaucoup de domaines comme la robotique mais aussi et surtout pour résoudre des problèmes d'optimisations. L'intérêt autour de cette méthode a grandi quand il a été montré, à partir de 2015, que l'apprentissage par renforcement permettait à une machine de gagner des parties sur des **jeux Atari** seulement en ayant connaissance des pixels de l'écran et du score de la partie. Plus récemment, **AlphaGo Zero** a été développé dans le but de jouer aux Go et est capable de progresser par lui-même. Ces deux exemples de jeu illustrent bien le concept d'apprentissage par renforcement, une machine progresse en ayant connaissance des informations du jeu (les pixels de l'écran ou l'état des pierres sur le plateau de go) et en recevant une récompense qui lui permet de savoir s'il joue bien ou non (le score sur Atari ou une récompense en fonction de sa position sur l'échiquier par rapport à celle de son adversaire).

Dans notre cas, l'apprentissage par renforcement va permettre à notre robot de se déplacer par lui-même pour atteindre une cible. Par ses capteurs, il connaîtra la distance aux obstacles, sa position en temps réel ainsi que celle de la cible, où encore sa vitesse actuelle. C'est en fonction de tous ces paramètres qu'une récompense sera calculée pour savoir s'il atteint ou non son objectif. Pour ce projet, nous utiliserons la bibliothèque **stable_baselines3** en **python** qui permet de réaliser de l'apprentissage par renforcement sur **des environnements gym** (cf II.3).

II.2 - Vocabulaire de l'apprentissage par renforcement

Avant d'entrer dans les détails techniques des modèles que nous allons mettre en place, il est nécessaire de détailler le vocabulaire de l'apprentissage par renforcement. En effet, les mots ci-dessous seront utilisés dans la suite de ce document pour décrire toutes les notions que l'on croisera. Pour chacun d'entre eux, on détaille sa correspondance avec le modèle actuel.

Agent : En apprentissage par renforcement, l'agent est l'objet qui va progresser au fil de l'apprentissage. Il peut être un robot ou encore un personnage de jeu vidéo.

Ici, l'agent sera notre robot dans l'environnement 2D et 3D.

Etat : L'état d'un environnement constitue l'ensemble des informations qui le caractérise à un instant t. On note l'ensemble des états S.

Actions : Ensemble de décisions que peut prendre un agent dans son environnement. On le note A.

Dans notre cas, les actions possibles sont au nombre de 4 : accélérer, décélérer, tourner à gauche ou tourner à droite (augmenter ou diminuer la valeur de l'angle formé par l'axe des abscisses et l'avant du robot)

Politique (Policy) : Fonction de l'ensemble des états vers l'ensemble des actions. Pour chaque état possible du système, la policy donne l'action que doit exécuter l'agent en retour.

La policy de notre modèle sera systématiquement un réseau de neurones dont la taille varie en fonction de l'algorithme utilisé. Celui-ci pourra être convolutif dans le cas où on utilise une caméra comme pour le modèle en 3D. À noter que la politique peut être une fonction probabiliste qui sera donc de la forme A x S -> [0,1], c'est-à-dire qui associe une probabilité d'effectuer une action a dans un état s donné.

Observation : Une observation est l'ensemble des informations que peut récupérer l'agent au contact de son environnement.

L'état et l'observation sont deux notions assez similaires, mais l'observation contient des informations moins riches que l'état. En effet, cette dernière ne contient que ce que l'agent peut observer. Par analogie, l'observation pourrait être décrite par la position de tous les objets qu'un observateur peut voir dans une pièce tandis que l'état serait la liste de toutes les positions de tous les objets de cette pièce. Cette distinction a une importance dans le cadre mathématique de l'apprentissage par renforcement. Cet apprentissage peut être décrit comme un processus de décision Markovien et pour cela il est nécessaire que l'état contienne l'ensemble des informations du système. En pratique, état et observation sont souvent confondus.

Dans notre cas, on décrira l'observation par un vecteur unidimensionnel, qui contient les informations suivantes :

- Position du robot : sa position sur les axes x et y
- Paramètres du robot : sa vitesse et son angle
- Position de la cible : sa position sur les axes x et y
- Distance aux obstacles : un nombre N de lasers tirés autour de lui afin de déterminer la distance aux objets dans ces N directions (afin de simuler un LIDAR)

Finalement, nous obtenons une observation de dimension 1x(6+N) et un espace d'action de dimension 4. C'est cette observation qui sera donnée en entrée de notre politique. **Ainsi, notre politique et un réseau de neurones à plusieurs couches, dont la couche d'entrée est constituée de 6+N neurones et la couche de sortie de 4 neurones.**

Récompense (Reward) : Valeur numérique qui permet de déterminer si un état est plus ou moins favorable à la réussite de la tâche

La reward est le point clé de l'apprentissage par renforcement, c'est elle qui va permettre à l'agent de savoir si ses actions ont un impact positif sur son objectif. Il est alors nécessaire de bien concevoir cette récompense. Le gros point noir de ce concept est qu'il n'y a pas de méthode systématique pour concevoir une reward. Il faudra donc **procéder de manière empirique** jusqu'à avoir les résultats les plus précis possible (voir partie II.6 pour plus de détail).

II.3 - Environnements d'apprentissage

Maintenant que le simulateur est en place et que l'on a déterminé la forme de l'observation et de l'espace des actions, il est nécessaire de mettre en place une interface qui fera le pont entre le simulateur et l'apprentissage : c'est **l'environnement**. En python, on utilisera **gymnasium** (anciennement **gym**) pour mettre en place une telle structure.

L'environnement est l'objet qui s'occupera, à chaque instant, de récupérer les informations provenant du simulateur pour les communiquer à l'algorithme d'apprentissage. En retour, il décrit toutes les modifications de l'environnement en réponse à une action à effectuer. Il calcule également la reward obtenue par l'agent dans un état donné. La figure II.3-1 illustre les interactions entre tous les éléments entrant en jeu en apprentissage par renforcement.

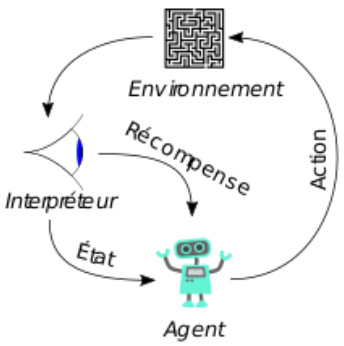


Figure II.3-1 : Interactions entre environnement et agent

L'environnement contient un nombre important d'informations lors de son initialisation qui permettent de faire fonctionner le processus d'apprentissage. On retrouve en particulier les dimensions de la fenêtre pygame, l'espace physique de pymunk, des références à des fonctions qui

gère la collision entre objets **mais surtout la définition des espaces d'observation et d'actions** dont voici l'écriture sur la figure II.3-2

```
self.observation_space = spaces.Box(low=-1, high=1, shape=(6 + self.nbRays,))
self.action_space = spaces.Discrete(4)
```

Figure II.3-2 : Définition des espaces d'observation et d'action

L'espace d'observation est une « Box », c'est-à-dire un espace continu dont les valeurs sont comprises entre les bornes low et high (ici -1 et 1). Sa taille vaut 6 + N, avec N le nombre de rayons du LIDAR, ce qui correspond bien à ce qu'on avait précisé sur l'observation dans le paragraphe II.2.

L'espace d'action est « Discrete(4) » ce qui signifie qu'une action est représenté par une **valeur entière comprise entre 0 et 3.** Pour chaque valeur, l'environnement la traitera pour effectuer des modifications sur les paramètres de l'agent.

Un environnement gym est composé de plusieurs fonctions qui vont être appelées successivement au cours de l'apprentissage. Voici un tour d'horizon des implémentations des différentes fonctions pour notre robot :

- **Fonction step() :** La fonction principale de l'environnement. Elle prend en paramètre l'action à effectuer par l'agent qui a été déterminée par la politique pour passer d'un état au suivant.

  Dans notre situation, on commence par actualiser les paramètres du robot selon l'action déterminée par la politique (on limite la vitesse de l'agent à 10 et son angle à +/- 180 degrés) :

Action	Effet sur l'agent
0	Vitesse + 0.5
1	Vitesse - 0.5
2	Angle + 5 degrés
3	Angle – 5 degrés

Tableau II.3-1 : Effet sur l'agent en fonction de l'action lors de l'appel à Step

  Puis à partir de ces deux paramètres, on actualise sa vélocité sur les axes de l'espaces. On appelle ensuite la fonction step() de pymunk sur l'espace des objets physiques pour les déplacer selon leur vélocité. Enfin, on appelle les fonctions get_observation(), reward() et terminated() pour renvoyer ces paramètres en sortie de fonction.

  On appelle timestep, un appel de la fonction step() de l'environnement. On appelle épisode, l'ensemble des états d'un environnement, de son démarrage jusqu'à l'appel de la fonction reset().

- **Fonction get_observation() :** C'est la fonction qui s'occupe de renvoyer les informations constituant l'observation sous la forme d'un vecteur unidimensionnel.

  Dans notre cas, l'observation est un vecteur de taille 6+N constitué de la position de l'agent, de la cible, des paramètres d'orientation et de vitesse de l'agent ainsi que les distances aux objets détectés par le N lasers du LIDAR.

- **Fonction terminated() :** Fonction qui détermine si l'environnement doit être remis à zéro. Cela est nécessaire quand l'agent atteint un état qui est beaucoup trop éloigné de son objectif et qu'il n'est pas pertinent de continuer l'apprentissage. Retourne un booléen qui vaut True si l'environnement doit être redémarré.

  Dans notre cas, on renvoie True si l'agent sort des limites de l'écran, si le nombre de timestep est trop élevé (ce qui veut dire que l'agent est trop long à accomplir sa tâche ou qu'il est bloqué) ou si le robot est entré en collision (obstacles ou sol dans l'environnement 3D)

- **Fonction reset() :** Fonction appelée lorsque terminated() renvoie True. Effectue une série d'instructions pour démarrer un nouvel épisode.

  Dans notre cas, la fonction reset() réinitialise les obstacles en supprimant les anciens pour en ajouter de nouveaux de manière aléatoire dans l'espace. On réinitialise ensuite la position du robot et de la cible en les plaçant aléatoirement dans l'espace.

- **Fonction reward() :** Fonction qui calcule la reward obtenue par l'agent dans un état donné. Elle est calculée à chaque timestep.

  Les détails de notre fonction de reward sont détaillés dans la partie II.6

Une fois l'environnement bien mis en place, la librairie stable_baselines3 qui permet de réaliser de l'apprentissage par renforcement, sera en mesure d'aller piocher les fonctions dont il a besoin pour entraîner l'agent. Des outils mis en place par cette librairie comme **la fonction env_checker(),** permet de vérifier que notre environnement est bien défini et propose des optimisations au besoin. Un point qui revient souvent dans les propositions d'optimisation de cet outil est **la normalisation des données**. Normaliser des données, c'est trouver une échelle commune pour toutes les données. Cela permet à l'apprentissage d'avoir des données de même ordre de grandeur, le contraire aurait pu poser des difficultés à traiter pour le réseau de neurones lorsque certaines informations peuvent être très petites et d'autres très grandes. Voici un récapitulatif des données normalisées dans le tableau II.3-2

Donnée	Espace initial	Espace normalisé
Position x : agent et cible	[-500, width + 500]	[-1, 1]
Position y : agent et cible	[-500, height + 500]	[-1, 1]
Vitesse	[-10, 10]	[-1, 1]
Angle	[-180, 180]	[-1, 1]
Distance obstacle	[0, 500]	[-1, 1]

Tableau II.3-2 : Échelle des données, avant et après normalisation

II.4 – Choix de l'algorithme d'apprentissage

L'environnement étant désormais en place, il est nécessaire de choisir l'algorithme qui s'occupera de l'apprentissage de notre modèle. Stable_baselines3 propose de nombreux algorithmes d'apprentissage qui présentent chacun leurs spécificités. En effet, il faut tout d'abord se renseigner sur la forme des espaces d'observation et d'actions qui sont tolérés par les algorithmes. Ci-dessous la figure II.4-1 résume les espaces d'actions acceptés par les différents algorithmes. Une même table existe pour les espaces d'observations.

Name	Box	Discrete	MultiDiscrete	MultiBinary
ARS [1]	✓	✓	✗	✗
A2C	✓	✓	✓	✓
DDPG	✓	✗	✗	✗
DQN	✗	✓	✗	✗
HER	✓	✓	✗	✗
PPO	✓	✓	✓	✓
QR-DQN [1]	✗	✓	✗	✗
SAC	✓	✗	✗	✗
TD3	✓	✗	✗	✗
TQC [1]	✓	✗	✗	✗
TRPO [1]	✓	✓	✓	✓
Maskable PPO [1]	✗	✓	✓	✓

[1] (1,2,3,4,5): Implemented in SB3 Contrib

Figure II.4-1 : Espace d'actions autorisé pour chaque algorithme d'apprentissage

Chaque algorithme utilise ses propres méthodes d'optimisation. Lors de ce projet, j'ai pu utiliser principalement deux d'entre eux que je vais détailler : DQN (Deep Q-Network) et PPO (Proximal Policy Optimization) et que j'ai pu mettre en place.

II.4.1 – Deep Q-Network

Space	Action	Observation
Discrete	✓	✓
Box	✗	✓
MultiDiscrete	✗	✓
MultiBinary	✗	✓
Dict	✗	✓

Figure II.4.1-1 : Espaces d'action et d'observations autorisés pour l'algorithme DQN

Afin de comprendre DQN, il faut d'abord s'intéresser **au Q-Learning**. Le Q-Learning est une méthode d'apprentissage par renforcement où l'agent cherche à apprendre une stratégie décrite par la fonction de valeur $Q: S\ x\ A \rightarrow R$. Celle-ci associe, pour un état et une action donnée, une valeur réelle. Ainsi l'agent choisi l'action $a \in A$ tel que $Q[s, a]\ soit\ maximale$. L'action optimale pour chaque état correspond à celle avec la plus grande récompense sur le long terme. Cette récompense est la somme pondérée de l'espérance mathématique des récompenses de chaque étape future à partir de l'état actuel. Au démarrage de l'algorithme, cette fonction est initialisée aléatoirement puis est réajustée au cours de l'apprentissage selon la formule suivante :

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

Avec :

- s et s', l'état précédent et l'état suivant
- a, l'action choisie
- α le taux d'apprentissage
- r la récompense reçue par l'agent
- γ le facteur d'actualisation (dont on reparlera dans la partie II.5)

Cette formule montre que la valeur de Q[s, a] est modifiée en une moyenne pondérée entre son ancienne valeur et le gain attendu. De cette manière, la fonction Q devient optimale pour répondre au problème posé.

DQN est donc une méthode basée sur le Q-Learning où la fonction Q est représentée par un réseau de neurones.

II.4.2 – Proximal Policy Optimization

Space	Action	Observation
Discrete	✓	✓
Box	✓	✓
MultiDiscrete	✓	✓
MultiBinary	✓	✓
Dict	✗	✓

Figure II.4.2-1 : Espaces d'action et d'observation autorisés pour l'algorithme PPO

PPO est un algorithme récent publié en 2017 par OpenAI. Il a été reconnu pour être très performant au point de concurrencer les algorithmes existants. Il est une alternative à DQN dans la mesure où son espace d'actions peut être continu.

C'est un algorithme plus complexe faisant intervenir des notions de statistiques mais dont l'idée est de maximiser la somme des récompenses sur le long terme : $R = \sum_{timestep} \gamma^t \, r(t)$ avec r la récompense au timestep numéro t et gamma le facteur d'actualisation. C'est une méthode policy-gradient, qui consiste en un modification des paramètres du système (ici les poids du réseau) à l'aide du gradient de la fonction d'objectif. Lors de l'apprentissage, PPO compare la politique actuelle avec la nouvelle actualisée. La fonction d'objectif contient deux termes principaux :

- Le terme de perte sur les politiques (policy loss) : Ce terme mesure à quel point la nouvelle politique diffère de l'ancienne. Il est conçu pour limiter les mises à jour de la politique à un petit pas (proche de l'ancienne politique) afin d'éviter des changements drastiques. La perte de politique peut être calculée à l'aide de diverses métriques, telles que la divergence de Kullback-Leibler (KL) entre les distributions de probabilité des actions de la nouvelle politique et de l'ancienne politique.

- Le terme de pénalité de rapport (clip ratio penalty) : Ce terme encourage la mise à jour de la politique dans la direction qui améliore les performances de l'agent. Il utilise un ratio entre les probabilités d'actions de la nouvelle politique et de l'ancienne politique pour ajuster les mises à jour. Ce ratio est ensuite limité à un intervalle spécifié (par exemple, [1 - epsilon, 1 + epsilon]) pour éviter des mises à jour trop importantes. La pénalité de rapport est définie comme la valeur minimale entre le ratio ajusté et le ratio non ajusté multiplié par l'avantage estimé de l'action.

II.5 - Optimisation des hyperparamètres

Lors des mes tests, j'ai pu me rendre compte au départ que les algorithmes avaient tendance à converger assez lentement (En termes de nombre de timesteps). Pour optimiser cela il est nécessaire de bien choisir les hyperparamètres de l'algorithme d'apprentissage.

Hyperparamètre : Paramètre externe qui influence le comportement et les performances d'un modèle d'apprentissage automatique, mais qui n'est pas appris à partir des données. Il doit être défini avant le processus d'apprentissage et affecte comment le modèle apprend et se comporte.

Il existe de nombreux hyperparamètres que l'utilisateur peut paramétrer à sa guise. Voici trois principaux paramètres que j'ai pu aborder tout au long de mes tests :

- **Learning rate (pas d'apprentissage) :** L'un des paramètres les plus importants du modèle. C'est lui qui dirige la vitesse de la convergence. Un pas trop petit entraîne un apprentissage lent mais augmente les chances d'atteindre un optimum de la politique. Un pas plus grand peut entraîner une instabilité lors de l'apprentissage et peut même entraîner une divergence du système. Cependant, cela permet également d'augmenter la vitesse de l'apprentissage.
  Il est donc important de bien régler cette valeur afin d'obtenir un apprentissage optimal. Lors de mes tests, j'ai souvent considéré une valeur d'apprentissage à 0.0003 qui est la valeur par défaut des algorithmes. Toutefois, il m'est arrivé pour certains d'entre eux de choisir une valeur adaptée linéairement, c'est-à-dire qui diminue au cours de l'apprentissage. Cela peut être utile pour éviter des optimums locaux en début d'apprentissage, puis d'affiner par la suite pour atteindre l'optimum global de la politique

- **Gamma discount factor (facteur d'actualisation) :** Le facteur d'actualisation est un hyperparamètre important dans la mesure où il est directement lié à ce que cherche à maximiser l'agent : la récompense. Gamma se situe généralement dans l'intervalle [0.9, 0.9997], et plus cette valeur est élevée plus les récompenses des timesteps tardifs d'un épisode seront prises en compte par l'algorithme d'apprentissage.
  Dans notre cas, la valeur de gamma se trouvera relativement élevée (au minimum à 0.99). En effet, nous verrons dans la partie suivante qu'il est important de donner une récompense très importante lorsque l'agent atteint la cible et qui est bien supérieure aux récompenses reçues à chaque timestep. Pour que cette récompense particulière soit bien prise en compte par l'algorithme, il est donc nécessaire d'avoir un gamma élevé car c'est la dernière récompense que recevra l'agent au cours d'un épisode.

- **Batch-size (taille de lot) :** Le batch-size est un paramètre qui détermine combien d'échantillons d'entraînement sont utilisés pour mettre à jour les poids d'un modèle à chaque itération d'apprentissage. Un batch-size plus grand accélère l'entraînement en effectuant des mises à jour moins fréquentes, tandis qu'un batch-size plus petit peut favoriser une meilleure généralisation.
  Pour ce paramètre, j'ai généralement opté pour une valeur assez faible (256 ou 512). Cela permet à l'algorithme d'actualiser plus souvent les poids du réseau de neurones. J'ai souvent pu remarquer qu'un batch size trop élevée va souvent entraîner des oscillations lors de l'apprentissage, alternant entre bonnes et mauvaises performances.

Afin d'optimiser ses paramètres, j'ai utilisé **la librairie python « optuna ».** Celle-ci permet de décrire un intervalle de valeurs pour chaque hyperparamètre, puis optuna s'occupe de tester sur différents apprentissages les paramètres pour obtenir les valeurs optimales qui maximisent la récompense de l'agent.

II.6 - Importance et choix de la reward

Passons au point clé de l'apprentissage par renforcement, mais aussi l'un des plus difficiles à mettre en place, **la fonction de récompense**. En effet, celle-ci dirige tout l'apprentissage en étant appelée à chaque timestep pour qualifier l'état dans lequel se trouve l'agent. C'est une **fonction empirique** qui est créée par le développeur et qui nécessite d'être recalibrée en fonction des performances de l'agent.

Il existe deux principaux types de fonction de récompenses :

- Celles qui sont **éparses**, c'est-à-dire qui récompensent l'agent après une succession d'actions (comme à chaque fin d'épisode par exemple). Elles permettent de laisser l'agent explorer son environnement jusqu'à être récompensé quand il trouve lui-même son objectif.
- Celles qui sont **denses**, c'est-à-dire qui récompensent l'agent à chaque timestep. Elles permettent de donner des indications régulières à l'agent pour qu'il sache s'il est dans la bonne voie pour atteindre son objectif.

Le choix d'un type de récompense plutôt qu'un autre dépend du problème que l'on étudie, de même que la forme de la fonction de récompense.

Pour nôtre modèle j'ai étudié plusieurs designs de fonctions de récompense :

- **Récompense binaire** : Lorsque j'ai commencé l'apprentissage par renforcement du robot, je me suis penché sur un système de récompense binaire, c'est-à-dire qui ne récompense l'agent qu'une fois que l'épisode est terminé et valant 1 s'il a atteint la cible, 0 sinon. Dans le cas où il n'y avait pas d'obstacles, cette récompense fonctionnait plutôt bien (même si assez longue à converger étant donné que le robot doit explorer tout l'environnement pour trouver la cible). Suite à l'entrainement, le robot réussissait à faire le lien entre ses variables d'observations (sa position et celle de la cible) pour atteindre son objectif. Cependant, une fois que des obstacles se dressaient sur la route du robot, ce type de récompense ne suffisait plus. L'agent a besoin de plus d'informations au cours de son apprentissage, doit-il rester proche des obstacles ? Doit-il s'en éloigner ? Quel chemin prendre pour se rapprocher de la cible ?
- **Récompense éparse avec obstacle** : Pour répondre à ces questions, j'ai commencé par détailler plus en profondeur la reward précédente. Je ne décerne toujours aucune récompense à chaque timestep, juste en fin d'épisode selon les critères suivants. On appelle dist, la distance entre l'agent et la cible.

Situation en fin d'épisode	Récompense obtenue
Nombre de timestep maximal atteint	-dist / 1000
Collision avec un obstacle	-1
Vitesse nulle	-1
Cible atteinte	(maxTimestep + 100 - nbStep) / maxTimestep

Tableau II.6-1 : Récompense en fonction de la situation

On divise la distance par 1000 dans le premier cas pour avoir une récompense supérieure à -1, en la laissant négative pour l'interpréter telle une pénalité. Dans le cas où la cible est atteinte, on donne une récompense qui sera plus ou moins grande en fonction du nombre de timestep qu'a mis l'agent à atteindre la cible. L'idée est de donner une récompense proche de 1 quand l'agent met environ 100 timestep pour réussir sa tâche. Même si cette récompense reste éparse, elle permet à l'agent de mieux appréhender les obstacles. Cela lui permet de faire le lien entre ses variables d'observation (distance aux obstacles grâce au LIDAR) et les pénalités qu'il reçoit en cas de collision. Ainsi, le robot réussit à atteindre la cible plus fréquemment qu'avant ; l'apprentissage reste toutefois lent.

- **Récompense dense avec obstacles v1 :** Pour la version suivante, je me suis penché sur une fonction de reward dense, où chaque timestep est une occasion pour l'agent de comprendre la situation dans laquelle il se trouve.

  La reward se construit de la façon suivante :

$$R(t) = \frac{1}{maxTimestep}\left(R_{speed}(t) + 2 * R_{obs}(t) + 3 * R_{dist}(t)\right)$$

  Avec $R_{speed}(t), R_{obs}(t)$ et $R_{dist}(t)$, 3 récompenses intermédiaires définies de la façon suivante au timestep t :

  - $R_{speed}(t) = abs\big(v(t)\big)$ avec v la vitesse du robot au timestep t.
  - $R_{obs}(t) = \frac{1}{n_l}\sum_{l=1}^{n_l} distObs_l(t)$ avec n_l le nombre de laser du LIDAR de l'agent et $distObs_l$ la distance à l'obstacle observée par le laser l (en ne conservant que les lasers ayant une vision sur un obstacle)
  - $R_{dist}(t) = \frac{-dist}{500}$ avec dist, la distance entre la cible et l'agent.

  On décide ici de pondérer les récompenses. Ainsi, on donne plus d'importance au fait d'éviter les obstacles, et encore plus au fait de se rapprocher de la cible. En définissant la reward de cette façon, l'agent reçoit à chaque timestep une récompense comprise entre 0 et 6/maxtimestep. Donc sur un épisode de longueur maximale, il aura accumulé entre 0 et 6 de récompense. On ajoute une récompense supplémentaire en fin d'épisode selon les critères suivants :

Situation en fin d'épisode	Récompense obtenue
Collision avec un obstacle	-10
Cible atteinte	10

Tableau II.6-2 : Récompense en fin d'épisode pour la récompense dense v1

  Nous avons ainsi défini une récompense plus complète qui permet à l'agent de mieux comprendre les obstacles qui l'entourent. L'apprentissage se fait de manière plus rapide, mais certains cas se trouvent problématique. Imaginons que l'agent soit entouré d'obstacles, alors aller dans une direction où une autre ne changera pas en moyenne la

reward, et il ne sera pas possible pour l'agent de comprendre quelle direction est optimale pour s'éloigner des obstacles.

Récompense dense avec obstacles v2 : Ainsi, pour pallier ce problème, on adapte la sous-reward d'obstacle pour qu'elle ne prenne en compte que les 5 lasers ayant les valeurs les plus faibles. Cela pousse l'agent à se concentrer sur l'obstacle le plus proche et pas sur tous les obstacles dont il a la vision.

C'est grâce à cette reward que j'ai pu obtenir le modèle le plus abouti. Maintenant que nous avons décortiqué le fonctionnement de l'apprentissage par renforcement, intéressons-nous aux résultats de tout ce travail.

Partie III - Étude des résultats

III.1 - Calculs effectués à distance

Tout d'abord, il faut savoir que l'apprentissage de tels modèles prend plusieurs heures. Même si le code a été optimisé pour être le plus rapide possible, le nombre de timestep par seconde calculé sur les ordinateurs que j'ai pu utiliser varie entre 1000 et 3000 selon le nombre d'obstacles et de lasers du LIDAR de l'agent. Cela peut paraitre beaucoup, mais plusieurs **centaines de millions de timestep** sont nécessaires pour atteindre la convergence de l'apprentissage. Les ordinateurs du techlab de l'école, certes un peu plus performant, ne permettait pas de dépasser de beaucoup l'intervalle décrit précédemment. Le problème du calcul de l'apprentissage est un problème de temps plutôt que de puissance de calcul.

Ainsi, j'ai décidé de mettre mon code sur un serveur distant qui s'occuperait de faire les calculs 24 heures sur 24. Celui-ci est doté d'un Intel Xeon E3-1230v6 et de 16GB de RAM DDR4. La connexion se fait en SSH via WinSCP sur lequel on peut démarrer une console pour contrôler le serveur.

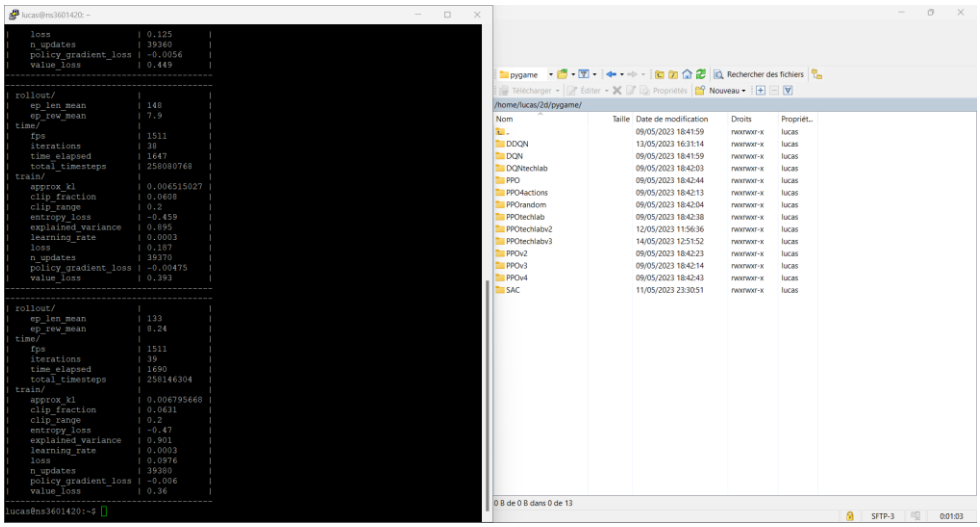
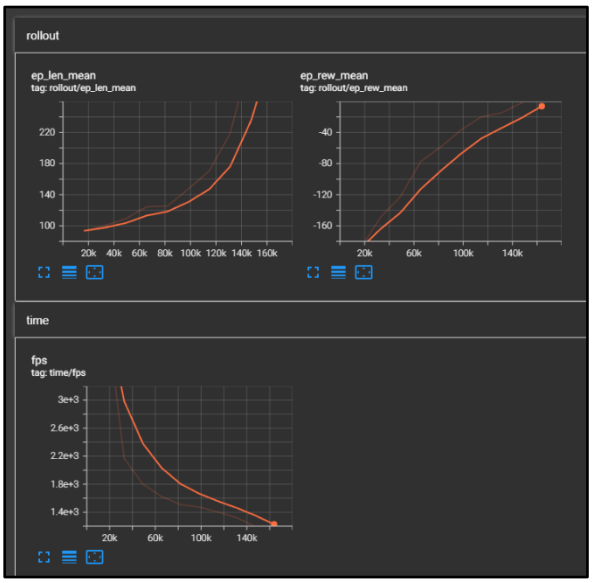
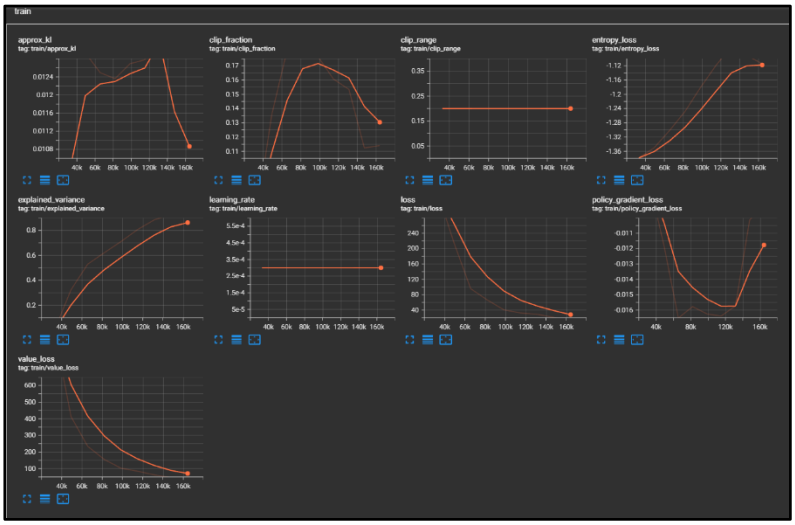


Figure III.1-1 : Interface de WinSCP

III.2 - Affichage des résultats avec tensorboard

Lors de la phase d'apprentissage, un ensemble de valeurs peut être aperçu en console. Celles-ci décrivent le déroulement du processus d'apprentissage. Il est important de bien les observer pour savoir si l'agent apprend à réaliser la tâche qui lui a été confiée. Pour se faire, on utilise **tensorboard** lors de l'apprentissage. Cet outil est proposé par tensorflow (spécialisé dans la thématique du machine learning sur Python) et permet de visualiser sous forme de courbe les données d'apprentissage en temps réel. Ce dernier enregistre dans un dossier toutes ces valeurs au fur et à mesure de l'apprentissage. La figure III.2-1 et III.2-2 illustre l'interface de tensorboard sur un exemple simple (l'environnement lunar_lander de gym, entrainé par PPO sur un peu plus de 150000 timesteps).





Figures III.2-1 et III.2-2 : Graphes des différentes valeurs obtenues lors d'un apprentissage par PPO

Le nombre de courbes dépend de l'algorithme d'apprentissage utilisé. PPO est l'un des algorithmes qui présente le plus de courbes lors de son utilisation. Chaque courbe décrit une valeur en fonction du nombre de timestep écoulé. Parmi ces données, on retrouve principalement :

- **Ep_len_mean et ep_rew_mean :** Respectivement la durée moyenne d'un épisode et la récompense moyenne reçue par l'agent lors d'un épisode
- **Explained_variance :** Détermine la qualité des estimations faites par le politique. Une valeur proche de 1 indique une bonne estimation des récompenses futures, alors que proche de 0 signifie qu'elle a du mal à prédire les récompenses.
- **Approx_kl :** détermine la mesure de divergence entre l'ancienne et la nouvelle politique ; à quel point la politique a évolué.
- **Value_loss :** Indique la valeur de la fonction de perte lors de l'apprentissage. Une value_loss élevée peut indiquer que la fonction de valeur a du mal à estimer correctement les récompenses futures.

Lors d'un apprentissage long, ce sont des milliers de points qui sont placés sur chaque courbes. Les valeurs obtenues peuvent être très variable d'un enregistrement à un autre. C'est pourquoi il peut être utile de lisser les courbes pour en dégager une tendance plutôt qu'estimer une valeur à un instant t. Dans le paragraphe suivant, on précisera les valeurs de lissage.

III.3 – Résultats d'apprentissage

Regardons désormais les résultats obtenus lors des différents apprentissages effectués. Pour la suite, on utilisera uniquement l'algorithme PPO et on comparera les résultats obtenus par le modèle de récompense éparse et le dernier modèle dense.

Tout d'abord au niveau de l'optimisation des hyperparamètres. A l'aide de la bibliothèque optuna, j'ai pu tester différentes valeurs du batch_size et du gamma. Par contrainte de temps, je me

suis limité à ces deux variables. Sur 33 tests de couples de valeurs, on obtient les résultats suivant sur la figure III.3-1 :

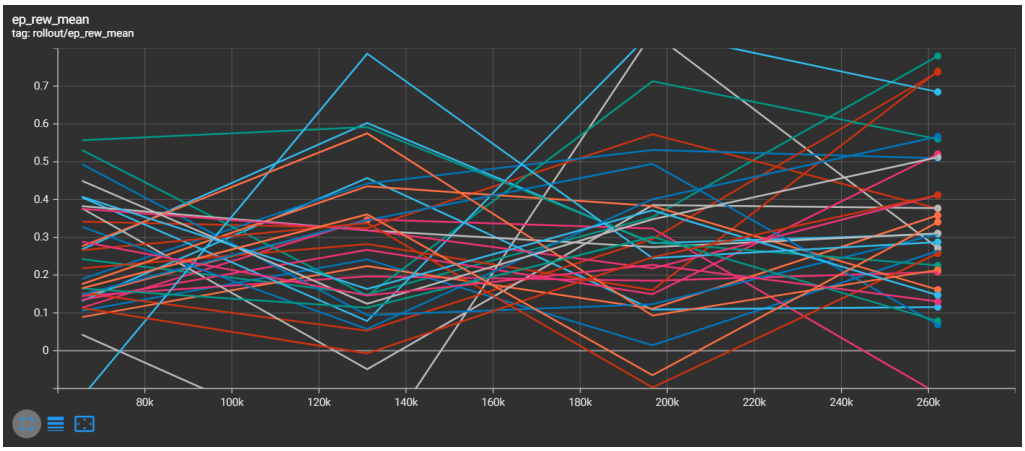


Figure III.1 : Optimisation des hyperparamètre gamma et batch_size sur 33 tests différents. (courbes non lissées)

Les valeurs des hyperparamètres n'apparaissent pas sur les courbes si dessus, mais on peut trouver des similitudes dans les résultats donnés par optuna. Il ressort que les courbes ayant des récompenses plus élevées en général, ont un batch_size plus faible que les autres courbes. Pour gamma, une valeur proche de 0.99 semble adaptée pour l'apprentissage.

Ainsi, pour les modèles qui suivront, nous fixerons la valeur du batch_size à 512, et la valeur de gamma à 0.991.

III.3.1 : Modèle éparse

Pour ce modèle, nous suivons le schéma de récompense décrit dans la partie II.6 et intitulé « Modèle éparse avec obstacles ». Voici les données obtenues affichées par tensorboard sur la figure III.3.1-1 :

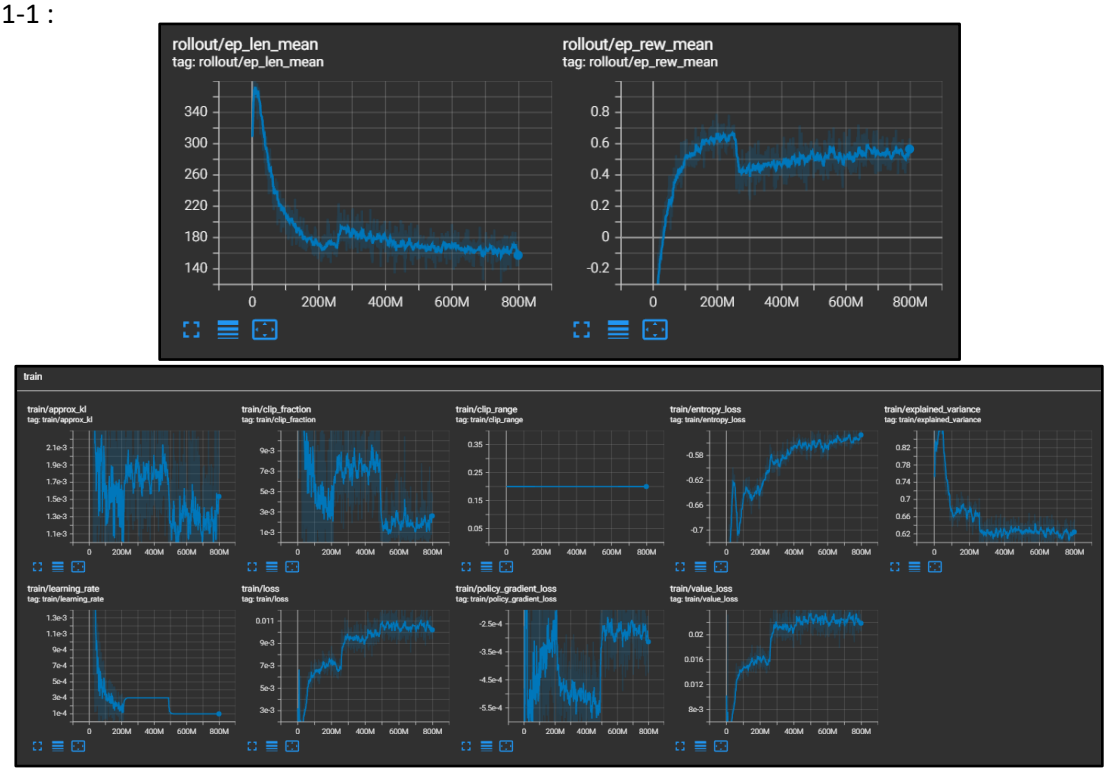


Figure III.3.1- 1 : Données d'apprentissage pour le modèle éparse (courbes lissées à 0.85)

Dans ce modèle, le seul moyen d'obtenir une récompense positive est de réussir à atteindre la cible. Quand l'agent y parvient, il reçoit une récompense inversement proportionnelle au temps qu'il a mis pour accomplir sa tâche. La récompense est calculée de telle sorte qu'il reçoive 1 point s'il parvient à la cible en 100 timesteps. (0.8 s'il y parvient en 200 timesteps)

Après avoir observé l'agent réaliser plusieurs fois la tâche, on peut estimer qu'il faut entre 80 et 150 timesteps pour atteindre son objectif. Ainsi, pour un modèle parfait où il réussit à l'atteindre à tous les coups, on peut s'attendre à une récompense aux alentours de 1. Dans notre cas, la courbe de récompense se découpe en deux parties. Dans la première (de 0 à 250M de timesteps) on entraine l'agent avec peu d'obstacles sur le terrain. Ensuite on entraine l'agent sur 5 obstacles.

Dans la première partie de l'apprentissage, l'agent commence à converger aux alentours de 0.7 points de récompenses. Une fois qu'on met tous les obstacles, la récompense moyenne descend vers 0.6 points. Ce qui signifie qu'il atteint une réussite moyenne de 60%. L'agent réussit à atteindre la cible un peu plus d'une fois sur 2, cela peut paraitre satisfaisant mais l'on peut faire mieux avec le modèle dense.

III.3.2 : Modèle dense

Pour ce modèle, nous suivons le schéma de récompense décrit dans la partie II.6 et intitulé « Modèle dense avec obstacles v3 ». Dans cette situation, je suis passé par 3 méthodes d'apprentissages différentes. Voici les données obtenues affichées par tensorboard sur la figure III.3.2-1.

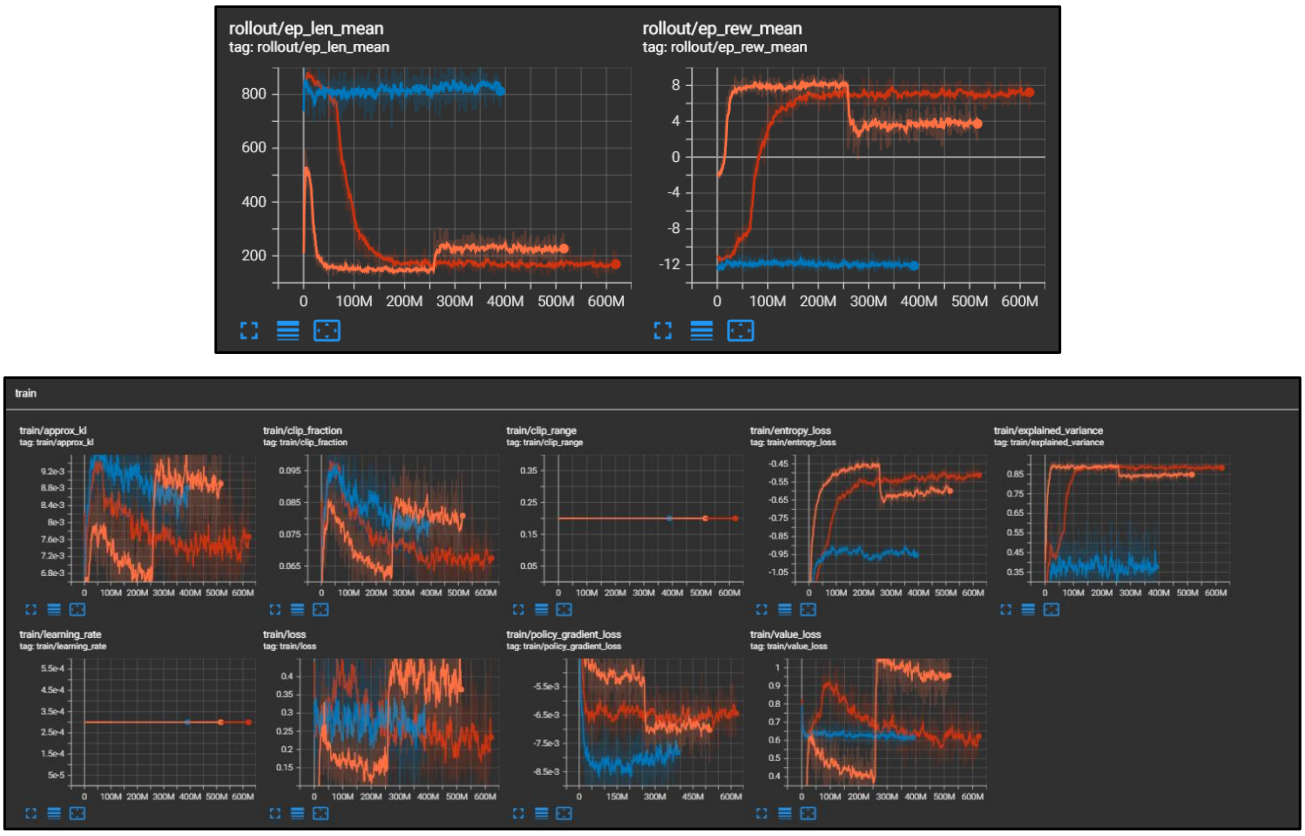


Figure III.3.2-1 : Données d'apprentissage pour le modèle dense (courbes lissées à 0.85)

Sur ces images, on peut observer 3 courbes différentes. La orange décrit un apprentissage à deux parties : de 0 à environ 250M de timesteps, on ne fait apparaitre que 2 obstacles dans le

simulateur puis on passe à 5 obstacles à partir de 250M. La bleue représente un apprentissage commençant directement à 5 obstacles à l'écran. La rouge provient d'un apprentissage où l'on génère entre 0 et 5 obstacles par reset de l'environnement. Ces environnements tournent à environ 1500 fps, soit environ 18h d'apprentissage pour 100 000 000 de timesteps.

Ces courbes montrent clairement que la manière dont on s'y prend pour l'apprentissage à un impact sur la capacité de l'agent à résoudre sa tâche. Dans le cas de la courbe bleue (5 obstacles dès le début), la reward est restée constante pendant les 400M de timesteps, soit près de 3 jours d'apprentissage, et donc l'agent ne parvient pas du tout à atteindre la cible. Pour la courbe orange, l'agent réussit à comprendre son environnement avec 2 obstacles autour de lui. Cependant, une fois que l'on passe à 5 obstacles, sa reward moyenne diminue (ce qui est logique étant donné qu'il est plus difficile d'atteindre la cible). Pour la courbe rouge, l'apprentissage est plus lent que dans le cas orange (200M de timesteps pour commencer à converger contre 50M dans le cas orange), pourtant la reward est plus élevée qu'à la fin de l'apprentissage.

Evidemment il est très difficile de comparer tous ces résultats qui dépendent chacun de paramètres d'apprentissage différents. Ici, il est clair que le cas bleu est hors-jeu, après 400M de timesteps, l'agent est incapable de progresser : lui donner un objectif trop difficile d'entrée de jeu n'aboutira à aucun résultat (ce qui est plutôt intuitif, réussir une tâche difficile sans avoir les bases est quasiment impossible). C'est pourquoi les cas orange et rouge sont plus pertinents. Dans le premier on entraine sur peu d'obstacles avant d'augmenter, dans le second on fait varier le nombre d'obstacles entre le minimum et le maximum. Concernant les autres courbes, l'explained_variance semble plus élevé pour la courbe rouge, la fonction de perte plus élevée pour la courbe orange. On pourrait penser que le résultat rouge est meilleur, pourtant la difficulté du problème est plus élevée dans le cas orange (5 obstacles en permanence). Il faut donc les comparer sur une échelle de difficulté similaire. Pour cela, on va les placer chacun dans un environnement à 4 obstacles et on va les tester sur 1000 épisodes. Celui qui obtiendra la plus grande moyenne de récompense sur ces 1000 épisodes pourra être considéré comme le plus performant.

Finalement, le modèle de la courbe orange a obtenu une récompense moyenne de 5.72 points et le rouge de 6.86 points. On peut donc considérer le modèle de la courbe rouge comme le plus performant des modèles développés jusqu'à présent. Voici une vidéo de son exécution dans l'environnement : https://youtu.be/bJotKTeecww

Ce modèle parvient à atteindre l'objectif en évitant les obstacles. Cependant, on peut encore voir assez régulièrement certains problèmes. En effet, l'agent a du mal à exécuter certains déplacements sensibles qui demande un peu plus de minutie. De même, il reste bloqué dans certains cas quand il est entouré d'obstacles. Même si c'est un premier modèle fonctionnel qu'il peut être intéressant d'adapter à un robot réel, il reste encore une marge de progression à combler pour avoir un modèle infaillible.

Conclusion

Ce projet de deuxième année à pour moi été l'occasion de découvrir cette troisième branche de l'apprentissage automatique qu'est l'apprentissage par renforcement. Basée sur les concepts d'exploration et de récompense, elle permet à un agent d'apprendre à réaliser une tâche par lui-même par des mécaniques empiriques.

Pendant toute cette année, j'ai pu découvrir les concepts théoriques de cette méthode d'apprentissage au point d'être assez familier avec le sujet. C'est avec regret que je ne peux continuer de concevoir de nouveaux modèles dans le cadre de ce projet, faute de temps. En effet, chaque apprentissage prenait plusieurs jours et il est donc difficile de beaucoup expérimenter quand le temps est compté. D'autant plus que j'ai commencé l'année en tant que néophyte dans ce domaine et qu'il a fallu plusieurs semaines avant de pouvoir appréhender les concepts de l'apprentissage par renforcement. Il n'est pas difficile sur internet de trouver des papiers de recherches traitant de mécaniques similaires d'évitement d'obstacles en lien avec l'apprentissage par renforcement, c'est donc à ma petite échelle que j'ai essayé de mettre au point quelque chose le plus performant possible.

Malgré cela, nous avons pu obtenir un premier résultat d'apprentissage qu'il serait intéressant de passer au monde réel. Ce modèle serait parfaitement adapté pour un robot type TurtleBot. Ce dernier étant circulaire, se déplaçant d'une manière similaire à celle décrite dans notre environnement 2D et utilisant un LIDAR pour la gestion des obstacles. Cela serait l'étape suivante du travail qui a été accompli dans le cadre de ce projet.

Évidemment, il est encore possible d'améliorer les modèles que j'ai pu proposer. D'autres idées d'apprentissage pourraient être mises en place. Par exemple, faire un apprentissage segmenté (comme on l'a vu à la fin de ce document) mais en changeant totalement la fonction de récompense en cours d'apprentissage. Une fois que l'agent à acquis une compétence, alors pourquoi ne pas essayer de favoriser la récompense sur une autre capacité à acquérir ? On pourrait également essayer de travailler avec plusieurs réseaux en parallèle à la manière de AlphaGo pour déterminer la qualité d'un état et adapter l'exploration en fonction.

L'apprentissage par renforcement a été un sujet que j'ai trouvé fascinant et que je souhaite approfondir à l'avenir. Lors de mes études l'an prochain à l'École Polytechnique de Montréal, je compte me spécialiser dans les systèmes intelligents et notamment en choisissant le cours d'apprentissage par renforcement. Cela me permettra de poser un cadre plus rigoureux autour des concepts que j'ai pu découvrir et d'éclaircir les zones d'ombres qui subsistent encore.

Bibliographie

[1] **CHOI, Jaewan, LEE, Geonhee et LEE, Chibum, 2021**. Reinforcement learning-based dynamic obstacle avoidance and integration of path planning. *Intelligent Service Robotics*. 1 novembre 2021. Vol. 14, n° 5, pp. 663-677. DOI 10.1007/s11370-021-00387-2.

[2] **Documentation | Bullet Real-Time Physics Simulation.** [en ligne]. Disponible à l'adresse : https://pybullet.org/wordpress/index.php/forum-2/

[3] **Gymnasium Documentation.** [en ligne]. Disponible à l'adresse : https://gymnasium.farama.org/index.html

[4] **OpenQuadruped/spot_mini_mini: Dynamics and Domain Randomized Gait Modulation with Bezier Curves for Sim-to-Real Legged Locomotion.** [en ligne]. Disponible à l'adresse : https://github.com/OpenQuadruped/spot_mini_mini

[5] **PHD, Wouter van Heeswijk, 2023. Proximal Policy Optimization (PPO) Explained.** *Medium*. [en ligne]. 31 janvier 2023. Disponible à l'adresse : https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b

[6] **Reinforcement learning, 2023. *Wikipedia*.** [en ligne]. Disponible à l'adresse : https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1155454334

[7] **Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.0.1+cu117 documentation.** [en ligne]. Disponible à l'adresse : https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[8] ***RL Baselines3 Zoo: A Training Framework for Stable Baselines3 Reinforcement Learning Agents*, 2023.** [en ligne]. Python. DLR-RM. Disponible à l'adresse: https://github.com/DLR-RM/rl-baselines3-zoo

[9] **Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations — Stable Baselines3 2.0.0a11 documentation.** [en ligne]. Disponible à l'adresse : https://stable-baselines3.readthedocs.io/en/master/

[10] **WENZEL, Patrick, SCHÖN, Torsten, LEAL-TAIXÉ, Laura et CREMERS, Daniel, 2021.** *Vision-Based Mobile Robotics Obstacle Avoidance With Deep Reinforcement Learning* [en ligne]. 8 mars 2021. arXiv. arXiv:2103.04727.