

“ Welcome back Mr Navigator ”

**0 - Introduction et objectif du projet**

Créer un agent intelligent capable de naviguer entre plusieurs points de passage dans un ordre précis et dans un environnements définis, tout en évitant des obstacles. Il doit pouvoir être utilisé comme personnage non jouable afin de rendre les séquences de nos jeux vidéo plus vivante, paraissant moins scriptés.

De nos jours, lors des tests de jeux la presse pointe souvent du doigt l’intelligence artificielle dans les défauts des produits, trop prévisible, toujours pareil...

Aussi, il est recherché de créer un modèle noyau afin de l'exploiter par la suite de manières isolés, en faire un socle, qui sera repris pour des entraînements plus spécifiques à certaines situations.

L’agent doit pouvoir être réutilisable peu importe son environnement et ses caractéristiques physiques.

L’agent doit pouvoir évoluer sur le sol, dans un premier temps nous excluons la possibilité d’un agent qui peut voler, si on devait transposer au monde réel si l’agent vole alors il peut tomber et donc casser, dans un cadre de recherche on préféra limiter la casse et le coût pour commencer.

**1 - Présentation de l'outil utilisé et l’environnement de développement**

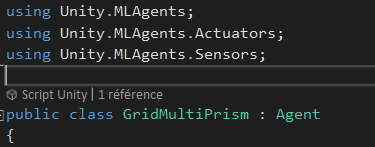
L’outil qui sera utilisé pour développer le prototype est le moteur 3D Unity, ce moteur est majoritairement utilisé pour le jeu vidéo et il est très populaire, il y a une immense communauté derrière. Depuis quelques années ils ont élargi leur spectre, ie moteur est maintenant utilisé pour réaliser des films d’animation et s’est modernisé en incluant de nombreux outils cloud et analyse de données entre autres, il y aurait énormément de choses a évoqué, l'outil est très puissant.

Unity fonctionne avec le langage C#, et fournit une API incroyable permettant d'interagir assez simplement assez toute une gamme de composant prés codé et la aussi redoutablement efficace. Ainsi de base tout scripts créés dans l'environnement dérive du MonoBehaviour :



L’architecture sous-jacente est .net 2.0 (celle utilisée dans ce projet, il y’en a d’autres mais en général on reste 2.0).

Pour pouvoir réaliser le projet nous allons utiliser la librairie ML Agents, ainsi nos scripts vont pouvoir dériver de Agent.



On va alors pouvoir utiliser les méthodes override de la librairie, qui va se charger d'établir toutes les connexions avec l'environnement python, ainsi on aura jamais à s'en occuper.



Le gros avantage c’est qu’on va pouvoir placer plusieurs environnement d'entraînement en même temps et donc considérablement accélérer le temps d'entraînement qui on le verra plus tard peut être très long, tous les agents seront connectées au même “Brain” et la librairie va traitée la tonne d’information qu’elle reçoit à la seconde et les synchronisés. Autre paramètre utilisé dans cette optique, l’espace temps, de base il est fixé sur 1 ce qui représente l’écoulement du temps réel.

Pour les sessions de train on peut l’augmenter (dans ce projet on alternera entre 15 et \* 20) ce qui revient à dire qu' 1 heure réel équivaut à 20h dans le moteur ce qui est un atout incroyable mais qui va demander une grosse machine, cette opération est très lourde, dans le projet on utilisera à la fin 4 agents donc 1h = 80h.

La limite devient la puissance de la machine, mais c’est très bluffant de constater que même à cette vitesse le framework le gère de manière identique, et n’est pas impactés par de faible performances.

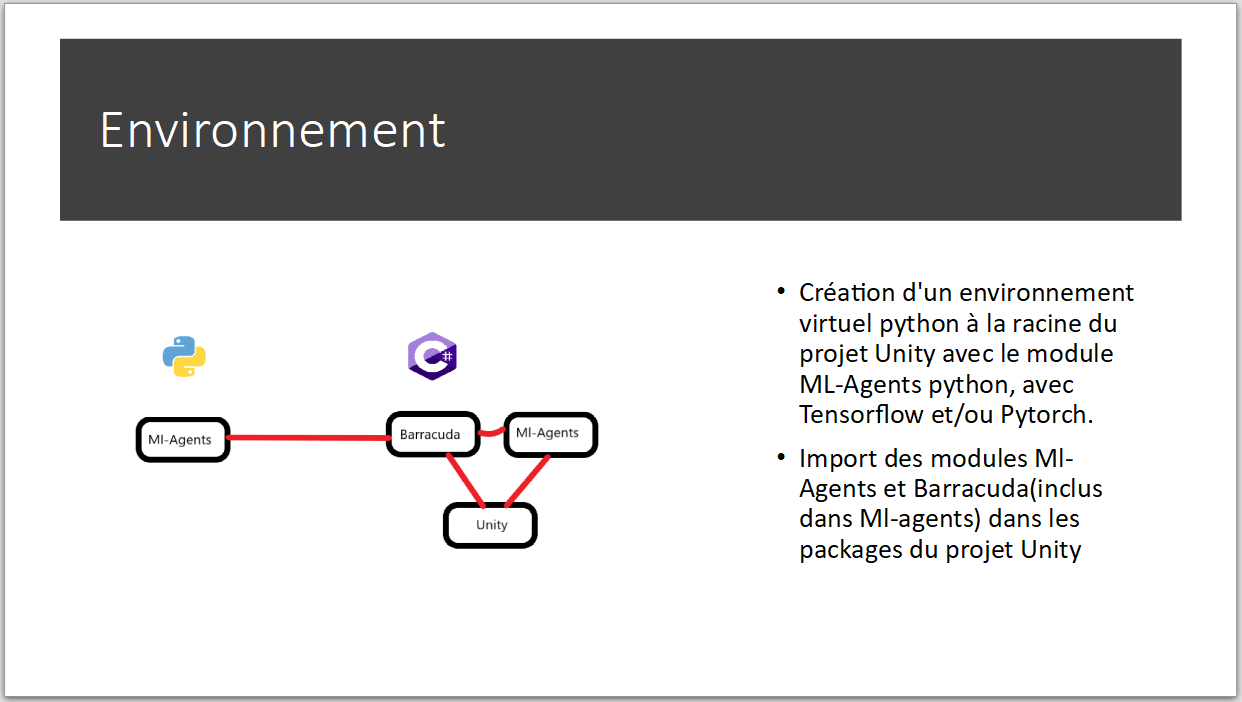
Si pendant l'entraînement on tourne à 10 frames par seconde ça suffit, en dessous j’ai constaté des problèmes, bien sûr l’idéal c’est 30 et +.

La version utilisée dans ce projet est la 1.6.0, pas l’officielle (celle du manager Unity j’ai directement pris celle du git pour avoir accès au grid sensor et aux exemples).

Plus d'informations sur le readme du dossier concernant les versions.

Voici le lien :

<https://github.com/Unity-Technologies/ml-agents/>



Quelques informations générales sur la libraire :

Début 2016/2017, V1.0 publiée en Mai 2020, mais le package reste en preview et n’est pas encore officiel pour Unity (ce qui signifie qu’elle s’adresse à des utilisateurs avertis en data science et en programmation aussi).

Les agents peuvent être formés à l'aide de l'apprentissage par renforcement, de l'imitation, de la neuro évolution ou d'autres méthodes d'apprentissage automatique via une API Python.

Le ML-Agents Toolkit est mutuellement bénéfique pour les développeurs de jeux et les chercheurs en IA car il fournit une plate-forme centrale où les progrès de l'IA peuvent être évalués sur les environnements riches d'Unity, puis rendus accessibles aux communautés de recherche et aux développeurs de jeux.

L'entraînement rend un objet .ONXX avec pytorch et peut donc être utlisée hors unity, on en reparlera un peu à la fin.

**2 - Présentation des concepts du reinforcement learning**

L’apprentissage par renforcement fonctionne selon un schéma de base commun, un agent est placé dans un environnement défini dans lequel il peut agir grâce à des inputs, par exemple un pressant un bouton il avance, sinon il bouge pas et c’est ce qu’il doit prédire, pour prendre ca décision il va observer son environnements, son état, et les récompenses qu’il reçoit.

Ce sont les rewards, une notion capitale pour la réussite d’un entraînement.

Son architecture est déterminante et doit être mûrement réfléchi, dans le principe ca parait simple mais il va falloir observer de près notre agent pour bien comprendre qu’il ne va pas forcément réfléchir comme nous car il n’aura pas tout nos sens.

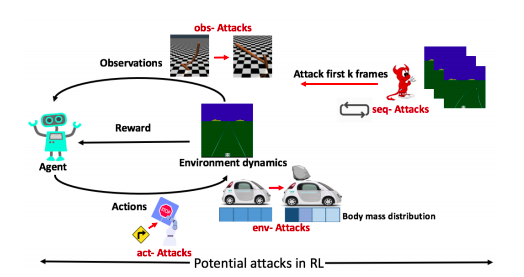
Si l’agent avance et qu’il est récompensé positivement il va apprendre à réaliser cette action en fonction de son environnement et idem pour les récompenses négatives.

Unity étant un moteur 3D on accède à la propriété physique de notre agent comme on peut en vrai avoir la vitesse d’une voiture, son poids, sa position, sa vélocité ect ect….

On peut le faire pour notre argent mais aussi pour tout objet dans l'environnement.

Si la voiture avance et qu’elle voit un véhicule statique en face d’elle, elle doit s'arrêter, si elle avance et alors je peux avancer car l’agent apprend qu’en gardant ses distances dans ces conditions il est récompensé.

Voici un schéma qui représente le cheminement pour agent qui pourrait se déplacer et avoir une action d’attaque possible.



**3 - Quelques notion de base relative à l’univers du jeux vidéo et nécessaire à la compréhension de la présentation**

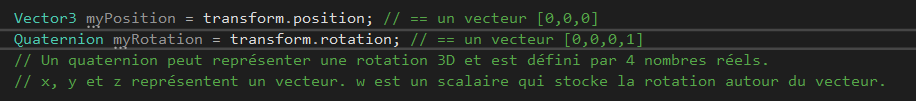
Dans ce projet il est important de comprendre les attributs que j’utilise comme features, je vais rester très en surface mais voici les 3 notions importantes:

* Transform

Chaque objet, même vide, a une propriété Transform qui représente sa position dans le monde par exemple si dessous un cube placé à l'origine du monde. Sur la droite on voit la position du centre de l’objet sur un axe X (largeur), un axe Z (profondeur) et un axe Y (hauteur).



Je peux accéder à ce composant avec le code sans même le référencer vu que c'est toujours présent.



Dans mon exemple ma variable myPosition est égale à (0,0,0), et idem pour la rotation sauf que c’est en réalité un vecteur de 4 données. On va pouvoir déplacer l’objet avec du code en agissant sur sa physique, dans le cadre d’un moteur 3D on évolue dans une sorte de boucle infinie qui est lue une fois par frame.

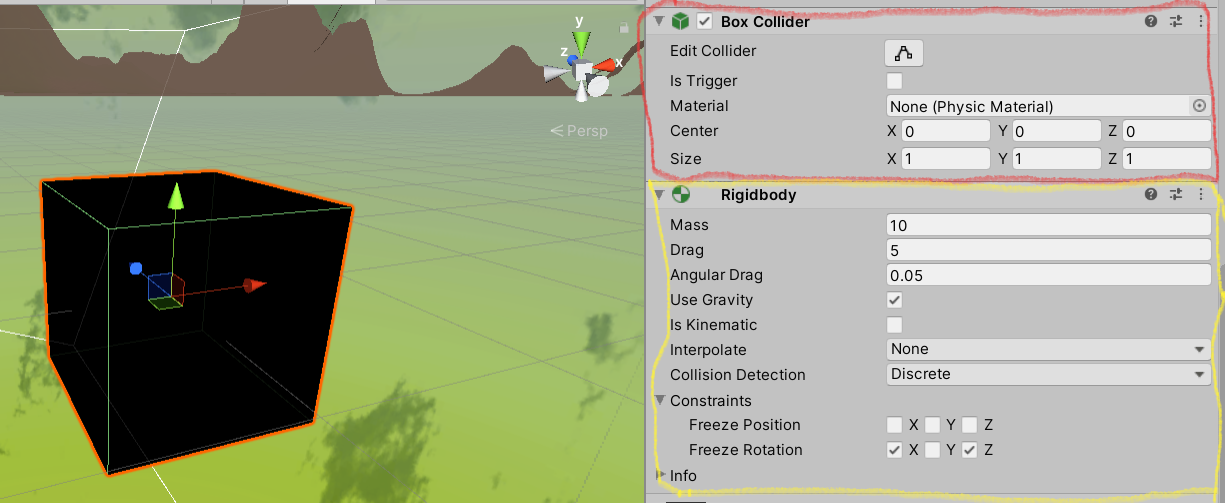
Pour être complet, le scale représente la grosseur de l’objet mais on ne joue pas avec cette propriété dans ce projet.

* Rigidbody

C’est le composant qui gère la physique, si il est absent notre objet n’en a pas et sera complètement statique. On peut ainsi fixer sa masse et la modifier en temps réel par exemple avec du code, toutes les propriétés afficher dans l’éditeur son accessible via le code et bien d’autres.

* Collider

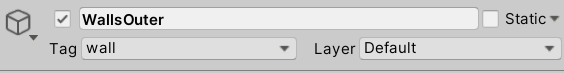
Représente l’espace qu’occupe l’objet, ces limites, un genre de couche invisible matérialisé dans l’éditeur par le contour vert



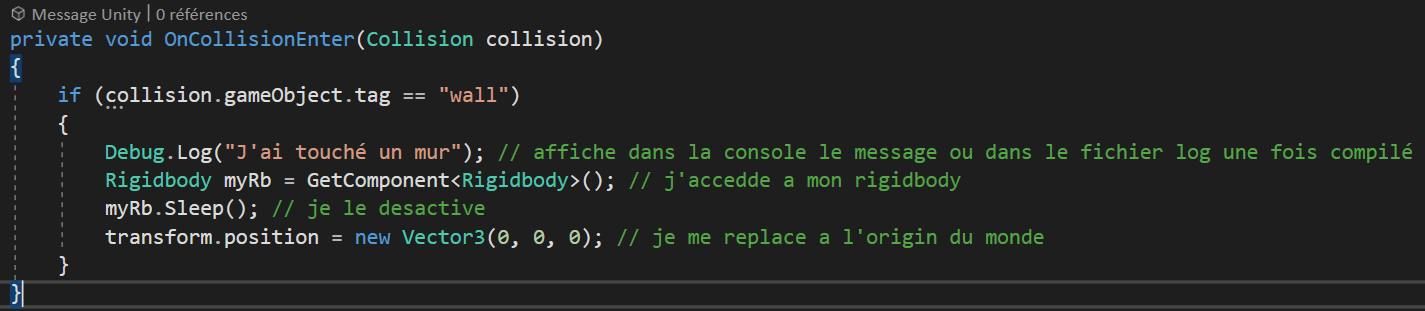
* Tag de l’objet et layer

Chaque objet peut être défini comme appartenant à un label ( tag) et peut être placé sur un layer, je ne jouerai pas avec les layers dans ce projet. Ce sont des genres de couches superposés on pourrait ignorer ainsi tous les objets sur un layer par exemple, ce qui laisse entrevoir les grandes possibilités de l'outil.

Si je place dans la scène ci-dessus un mur et que je lui attribue le tag wall, et que je le laisse sur le Layer par défaut comme mon cube.



Si j’exécute un script sur mon cube vous devez être mesure maintenant de comprendre le code suivant :



Si mon cube touche un objet dont le tag est wall, je désactive ma physique et je replace mon cube au centre du monde

Noter que pour les observations notre réseau de neurones utilisera des vecteurs normalisés.



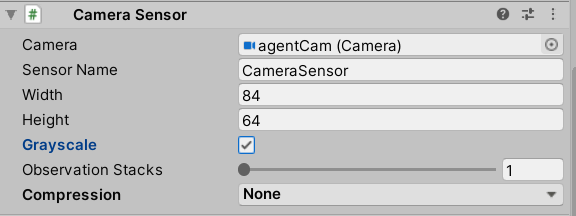
C’est aussi simple que ça et ça pourrait en inspirer d’autres ….

***3 - Acquisition des données et explication du choix de la méthode sélectionnée.***

Maintenant que les bases sont posées direction la data science :)

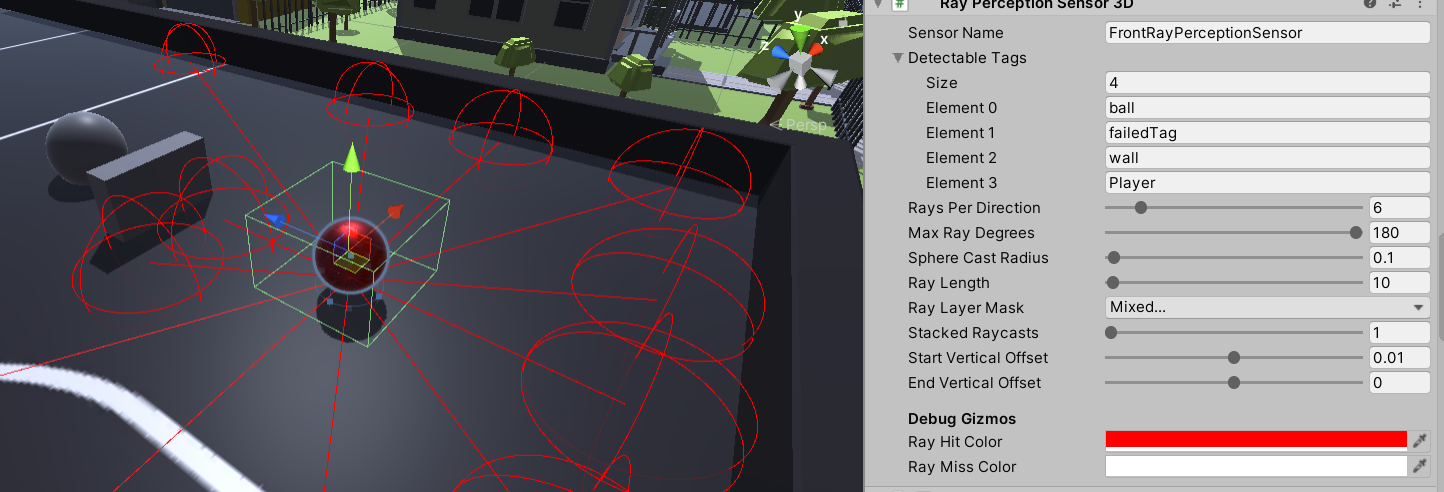
La librairie fournis 3 outils à ce jour, ils présentent tous des avantages et des inconvénients et je vais expliquer mon choix pour ce projet :

* La caméra : ce n’est ni plus ni moins que de la vision, c’est le plus simple à mettre en place, peu de réglage, par contre c’est très gourmand en ressources et sur mon pc ça n'était pas viable mon processeur été complètement saturé.



Le composant est fourni il reste plus qu'à le régler et c’est censé suffire si on estime que les informations visuelles sont des features suffisante pour notre modèle.

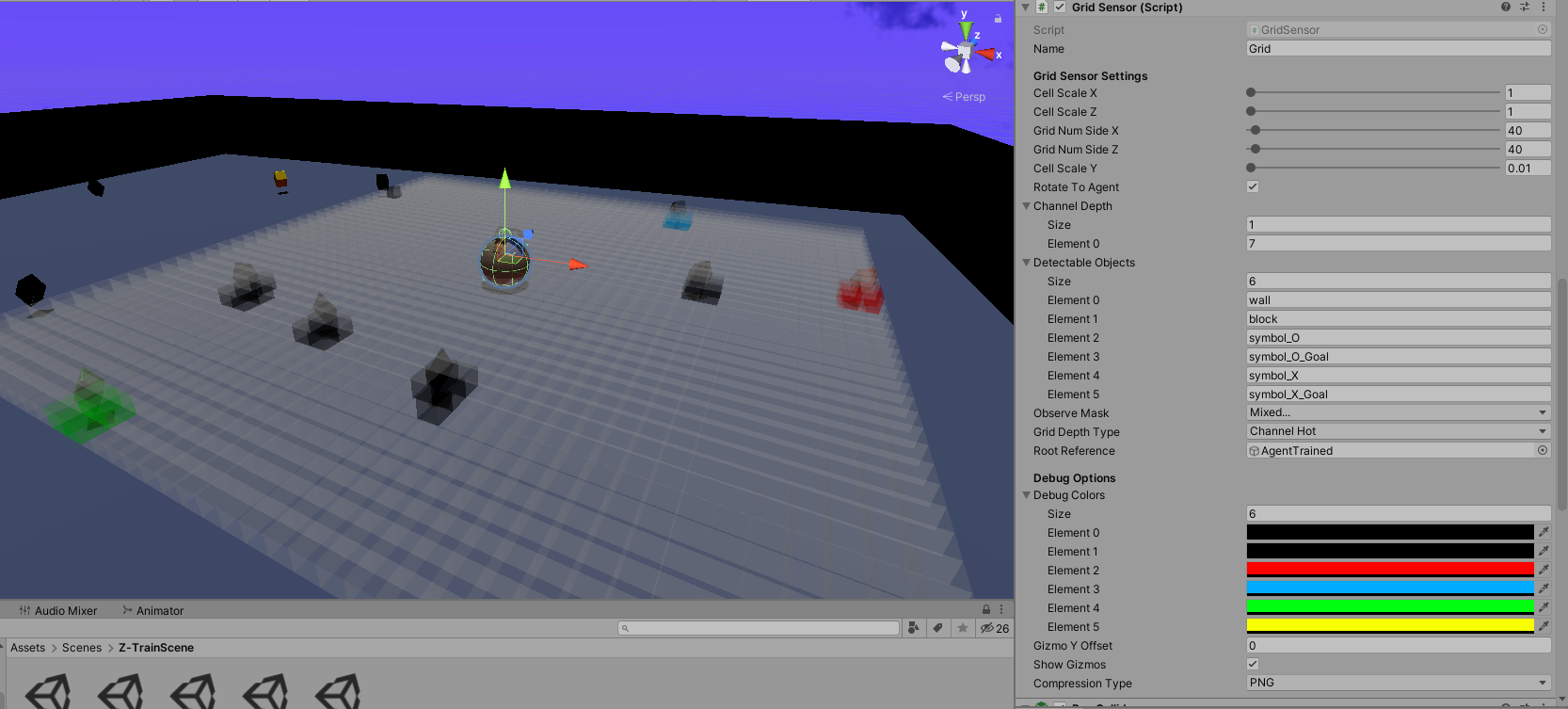
* Les Raycast : Méthode classique



Le composant est très efficace et c’est le mieux optimisé. On peut paramétrer nos rayons un peu comme on le veut, et préciser qu’elles sont les tags à observer, mais aussi les layers, on peut en mettre 1 rayon ou partout à 360 degrés. Ils renvoient des valeurs normalisées, par exemple la distance entre ma transform.position et le wall.transform.position. Le problème qu’on rencontre avec ces rayons c’est qui ne traverse pas les objets par exemple dans la figure les rayons bloquent contre le mur et l’agent ne voit plus la sphère derrière. Il se retrouve un peu bêta dans ces situations et ne sait plus trop quoi faire, et s’il perçoit un objet plus loin il partira ce qui n’est pas très logique et donne un rendu un peu bizarre, il va se retrouver à tourner sur lui-même par exemple pendant plusieurs secondes.

* Le Grid Sensor : Méthode retenus pour le projet :

Ce composant est très récent, dans le projet c’est la première version publiée que j’utilise.



Voici la première illustration pour notre projet. Notre agent est au centre de la grille que je vais paramétrer un peu comme les rayons, donné les tags à observées sauf qu’ici les tags sont associés à un code couleur. Ici l’objectif est passer les points dans l’ordre déterminé à savoir Rouge, Vert, Bleu, Jaune, sans jamais touché les Noirs.

Peu importe le tag ou sa représentation 3D, le grid sensor permet de généraliser la logique ce qui n’est pas le cas des rayons. De plus, notre problème de détection des objets est réglé ici, le point vert est bien perçu par l’agent malgré les obstacles qui sont entre eux.

Le Grid Sensor combine la généralité de l'extraction de données à partir de Raycasts avec la puissance de traitement d'image des réseaux de neurones à convolution. Le Grid Sensor peut être utilisé pour collecter des données sous la forme générale d'une matrice "Largeur x Hauteur x Canal" qui peut être utilisée pour la formation des agents d'apprentissage par renforcement ou pour l'analyse des données. Si vous voulez une explication approfondie sur ce point j’ai joint au dossier un fichier (Annexe 1 Grid-Sensor MD) issue de la documentation, et avec le package est fournis un exemple d’utilisation, c’est à partir de cet état de l’art que j’ai commencé mes travaux.

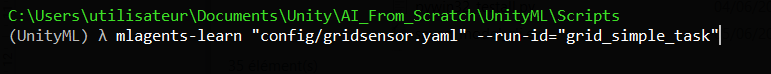
***4 - Etat de l’art et premier modèle***

Vous trouverez sur cette page de blog l'environnement fourni en exemple avec un extrait vidéo au milieu de la page.

[***https://blogs.unity3d.com/2020/11/20/how-eidos-montreal-created-grid-sensors-to-improve-observations-for-training-agents/***](https://blogs.unity3d.com/2020/11/20/how-eidos-montreal-created-grid-sensors-to-improve-observations-for-training-agents/)

Dans un premier temps pour la prise en main j’ai cherché à recréer l’exemple, j’ai donc copié l'architecture du réseau de neurone et les propriétés physiques de l’agent.

Il me reste (presque...) plus qu'à exécuter la ligne de commande suivante depuis mon environnement virtuel python et exécute l’application dans unity:



Vous trouverez (en Annexe 2) une capture vidéo de ce premier train ( si vous la téléchargez, vous aurez la définition Hd d'origine, elles sont compressées dans le cloud).

Cette vidéo a été prise environ 5/10 minutes après le début du train, l’agent fait un peu n’importe quoi, notre environnement python nous retourne des logs, on peut suivre l’apprentissage grâce à la moyenne des rewards obtenus, et on peut y accéder depuis une interface tensoard board en temps réel, il y’a d’autres métriques comme la loss, et la durée des épisodes en graphique.

Petite différence entre le cas d’origin et notre objectif, on veut une boucle qui a une fin, ce qu’eux n’ont pas fait, ils font apparaître les objets ainsi l’agent ne se retrouve jamais dans le cas où il reste 1 point positif et tous les négatifs, on verra plus tard que ce point un peu critique j’ai eu du mal à le gérer dans un premier temps.

Dans ce premier exemple l’agent est récompensé de 1 si il se déplace et touche un points rouge, et de -1 si il touche un mur ou un obstacle ce qui entraîne la fin de l’épisode et tout la scène est réinitialisé, dans le but d’attendre une généralisation on fait apparaître tout au hasard dans l'environnement y compris l’agent (pas les murs mais on pourrait).

Il y a 5 objets à ramasser, ainsi la fourchette de rewards qu’on trace ira de -1 à 5 si l'épisode est terminé avec succès.

Vous trouverez une nouvelle capture vidéo prise à un moment où l'agent a commencé à bien progresser. (Annexe 3)

Vous verrez aussi l’interface tensorboard, c’est ici que son logué les métriques de réseaux de neurones. On observe la courbe de rewards, la loss, on peut choisir d’y logué nos propres métriques comme un score ou la santé de l’agent par exemple.

On peut ainsi comparer nos différents run et voir si nos actions ont amélioré le modèle.

**5 - Création et Amélioration du modèle**

Vous trouverez dans le dossier 2 fichiers .cs ce sont les scripts C# qui permette de reproduire l’expérience, AreaPrisme est le script qui gère l'environnement de l’agent selon si il est en Train ou Inférence, et gère les informations des objets de l'environnement et GridMultiPrism est le code de l’agent.

Toutes les méthodes Override sont celle qui communique avec python on retrouve en générale dans l’ordre :

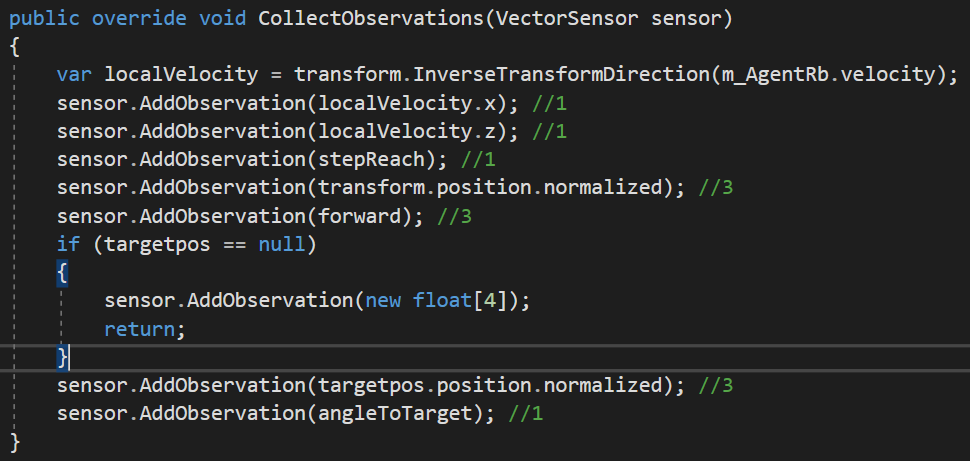
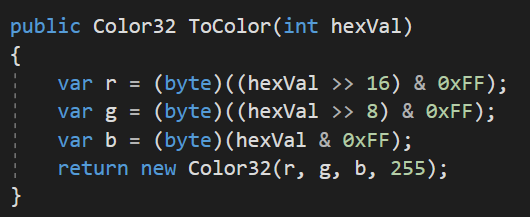
* Iniitialize () : Lue une seule fois au départ (comme awake et start, après le start).
* OnEpisodeBegin() : On nettoie la scène à chaque début de séquence, et on fait apparaître tous les objets au hasard, agent compris, dans l'environnement grâce au fonction du script AreaPrisme.
* Heristique() + OnActionReceived() (c’est ici qu’on gère les contrôles de l’agent, si il est pas en Train ou Inférence, on peut nous humain le controller)
* CollectObservations(), on peut rajouter des features en plus de la grille
* EndEpisode() méthode appelée pour terminer un épisode soit quand l’agent a fini sa tâche et j’aime bien aussi l’utiliser dès que l’agent se trompe, tous ne font pas ça.

Rappel : Notre modèle doit pouvoir se déplacer entre 4 point de passage dans un ordre précis toujours le même et pour ça il doit prédire ses actions qui sont dans ce projet discrétisé en 2 branche avec 3 possibilités

* si il appuie sur le bouton S il recule, Z il avance, ou il fait rien (=3)
* si il appuie sur Q il tourne sur la gauche, D la droite, ou rien (=3)

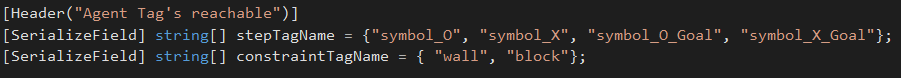
Ainsi il pourra avancer tout en se tournant, c’est ce qu’il doit prédire. (Méthode Heuristique du script agent).

Pour ce faire on a donc notre PNG la grille et ses couleurs + les observations en features.



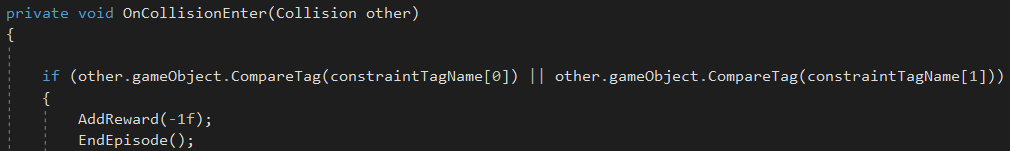
Je rajoute en observations la vélocité de ma mass sur l’axe X et Z, on ne se déplace pas sur Y, l’étape ou on est (un int , 0 au départ puis + 1 par étape), ma position, forward est un vecteur 3 qui donne la direction en face de l’agent et s'il y'a un objectif, sa position, et un angle c’est un DOT product, float entre 0 et 1, la valeur est à 1 si l’agent regarde la target. Pour cette action il est légèrement récompenser +0.001 par frame l’objectif été de le forcer à moins se déplacer en marche arrière.

Enfin il faut un pattern pour les Rewards, vous les trouverez aux lignes AddRewards(), Les principales se trouvent dans la fonction CheckSteps() de l’agent, elle est appelée à chaque collision, la fonction vérifie les informations en fonction de l’objet, l’étape en cours.



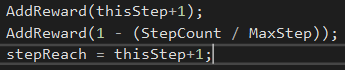
Ces 2 lignes de code, permettent au développeur d’avoir la main depuis son interface et donc mettre ce qu’il veut, le premier ce sont les tags positifs dans l'ordre, l’autre les tags contraintes.

Si il y’a collision on vérifie si c’est une contrainte



L’agent est pénalisé, l’épisode terminé.

SI c’est positif la fonction CheckSteps() attribuera :

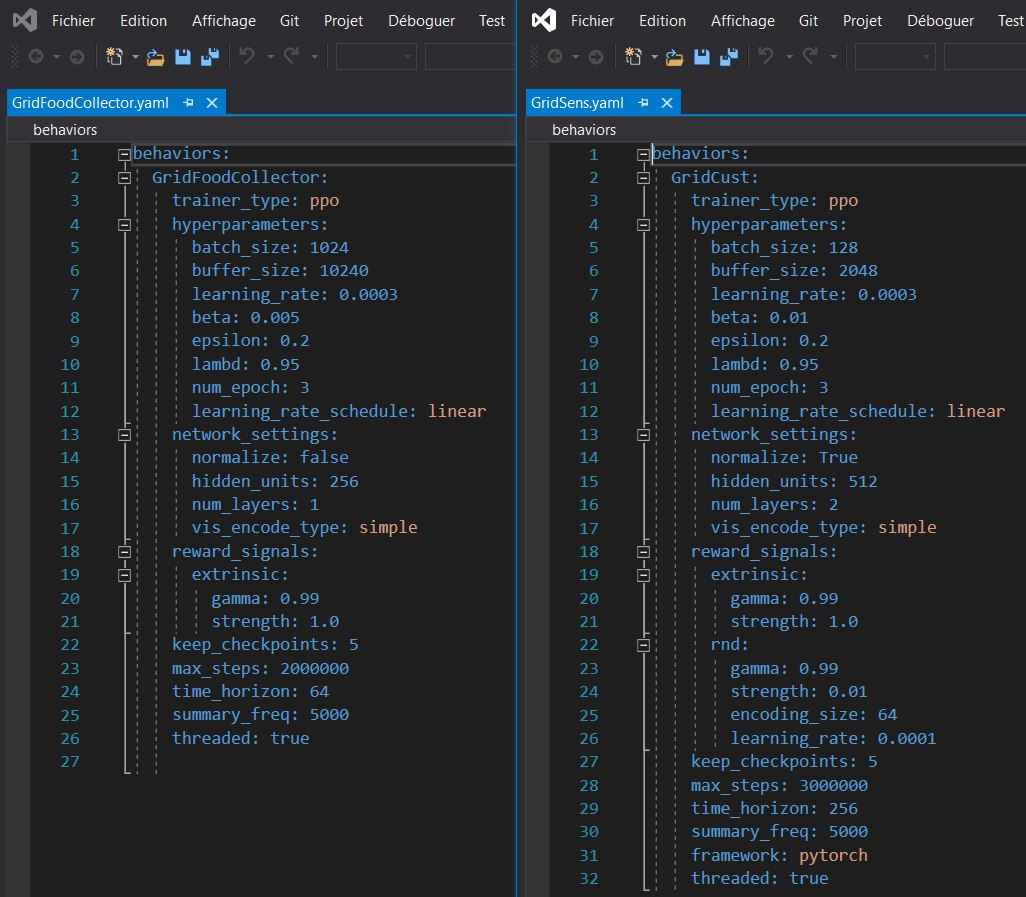


Noter que la 2ème ligne pénalise l’agent plus il est lent.

En réalité, notre agent doit comprendre cette fonction, sa logique pour savoir comment se déplacer.

Pour commencer l'entraînement sur notre projet je vais tester mon code avec le réseau de neurone paramétré à l’identique, venons d'ailleurs, l'algorithme utilisé est le Proximal Policy Optimization (ppo) et le framework Pytorch.

Voici les réglages de l'algorithme, à gauche celui de l’état de l’art, à droite celui qui m’a permis d’atteindre mon objectif.



J’ai noté mes observations au fur à mesure de mes itérations dans l’outils microsoft whiteboard, j’ai ouvert le lien au passage, je ne le détaillerai pas ici comment je suis arrivé à mes fins, mais je peux m’appuyer dessus lors de la présentation si besoin.

(L’outils a la mauvaise habitude de changer les couleurs des courbes si on supprime des entraînements, j’en ai fait les frais et ça rend le suivi un peu plus compliqué).

https://wbd.ms/share/v2/aHR0cHM6Ly93aGl0ZWJvYXJkLm1pY3Jvc29mdC5jb20vYXBpL3YxLjAvd2hpdGVib2FyZHMvcmVkZWVtL2RjNGQ2NDAyYmRmODRmMTdhYTViOGY3Y2NjMzM1OGYwX0JCQTcxNzYyLTEyRTAtNDJFMS1CMzI0LTVCMTMxRjQyNEUzRA==

J’ai augmenté les neurones de 256 à 512, j’ai augmenté la fenêtre de vision car dans notre cas, il faut que l’agent puisse se souvenir suffisamment longtemps de ces observations (time\_horizon : 256).

En tant que tel, ce paramètre fait un compromis entre une estimation de variance moins biaisée, mais plus élevée (horizon temporel long) et une estimation plus biaisée, mais moins variée (horizon temporel court). Dans les cas où il y a des récompenses fréquentes dans un épisode ou la mesure est d'une taille prohibitive, un nombre plus petit peut être plus idéal.

Ce nombre doit être suffisamment grand pour capturer tous les comportements importants dans une séquence d'actions d'un agent.

Cela correspond au nombre d'étapes d'expérience à collecter par l’agent avant de l'ajouter au tampon d'expérience. Lorsque cette limite est atteinte avant la fin d'un épisode, une estimation de valeur est utilisée pour prédire la récompense globale attendue de l'état actuel de l'agent.

J’ai aussi rajouté un layer qui correspond au nombre de couches cachées présentes après l'entrée d'observation ou après le codage CNN de l'observation visuelle. Pour des problèmes simples, avec moins de couches, l'entraînement est plus rapide et efficace.

Plus de couches peuvent être nécessaires pour des problèmes de contrôle plus complexes, comme on a 2 branches j’ai mis sur 2, il n’est pas recommandé d’aller à plus de 3.

J’ai mis normalize sur true, j’ai réduit le batch\_size et le buffer afin d’avoir un update du modèle plus rapide, et j’ai combiné les méthodes de rewards intrinsic and extrinsic, pour ça j’ai rajouté tout le bloc RND il s’agit de paramètre qui ajoute à l’agent une notion de curiosity. En substance, il encourage les agents à explorer l'environnement plus efficacement lorsque les récompenses sont peu fréquentes et peu distribuées. L’objectif est d'accélérer l'entraînement et dans certains cas un modèle plus performant, cela va dépendre de ce qu’on cherche à faire.

Plus d’explication sur le sujet :

https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/

Malgrés cela le modèle que vous verrez en démonstration a suivi un train d'environ 20h avec 4 agent en \* 20 soit environ 1600 heures, ce qui correspond à 66 jours réel. Bien entendu on n’est pas obligé de repartir de 0 à chaque fois on peut recharger son modèle ainsi une fois satisfait j’ai augmenté le nombre de contraintes (zone noire) une fois la logique apprise avec 1, j’ai rajoutée de la difficulté, je suis passé à 5, puis idem à 10. Si on avait commencé à 10 le train aurait sûrement été plus compliqué au départ.

(Annexe 4) une vidéo montrant la fin du train avec 1 seule contrainte.

(Annexe 5) Démo du modèle en inférence dans diverses scènes afin de tester la généralisation dans ces scènes il y’a entre 5 / 20 contraintes, parfois le modèle est sur un agent qui n’a plus les mêmes caractéristiques physiques. Pourtant il fait ce qu’il faut, c'est impressionnant, mais il sera moins précis dans ces déplacements.

**6 - Axe d’amélioration**

* Corriger le problème de réapparition sur un objet, dans ce cas l’agent est directement pénalisé et l’épisode enregistre un reward de -1, cela créer du bruit.
* Corriger la méthode pour que l’agent regarde enfin son objectif et se déplace moins en arrière, d’une façon générale améliorer son comportement.
* Pour être plus souple en inférence et dans la généralisation essayer d’agrandir le capteur ou plutôt la taille des cellules, dans les 2 cas l’agent couvrira une zone plus large, J’ai remarqué qu’il est très précis si il perçoit les 4 points dans la matrice, les observations de position sont juste un moyen d'accélérer l'entraînement, elles ne prennent pas le poid sur la matrice, ce qui donne des erreurs en inférence sur une zone plus grande et comportement hésitant.
* Un PC plus puissant pour mettre 100 agents pendant le Train (Please).

**7- Ouverture vers le monde réel.**

Évidemment le plus accessible pour exploiter ces IA c’est les jeux vidéo, ce qui sous-entend toute la gamme des mobiles aux casque VR/AR.

Mais les jeux vidéo sont parfois produits à but pédagogique voir même thérapeutique, pour ces domaines les IA sont présentes depuis longtemps et aujourd’hui ces méthodes ouvrent de nouvelles perspectives.

Aujourd’hui si on veut transposer des agents vers le réel on peut commencer par la réalité mixte.

Il existe un programme professionnel entre Azure et Unity à ce sujet, et j’ai les compétences pour mixer ces technologies.

On peut utiliser aussi toutes la gamme azure comme les cognitifs services, ce qui permettra d’amener nos IA ailleurs que dans une page web, c’est je pense l’avenir de L’IA, tout du moins on se rapproche, plus le temps passe, de concept seulement évoqué dans les films de science-fiction il y’a 20 ans.

Il y a tout un programme de tutoriel fourni par Microsoft à ce sujet, il me manque juste le matériel et le soutien financier.

<https://docs.microsoft.com/fr-fr/windows/mixed-reality/develop/unity/tutorials/mr-azure-307>

****

Unity nous donne la possibilité de faire de l’imitation learning et de compiler l'environnement de Train sur le support, comme suggéré l’illustration, imaginons utiliser un casque AR, qui apprend par imitation les tâches professionnelles d’une personne. Les possibilités sont nombreuses.

Pour aller plus loin, certaines personnes ont réussi à exécuter les modèles d’unity sur de vrais robots, alors le rendu est encore léger pour une utilisation tout public.

Ils ont utilisé les rayons, je ne sais pas vraiment si il est raisonnable de penser qu’outil comme grid Sensor existe ou puisse exister un jour.

Voici une vidéo en exemple :

<https://www.youtube.com/watch?v=sFY_0e1nTQE&t=115s&ab_channel=JuhaS%C3%A4lli>

Ils ont aussi publié un GitHub ou ils expliquent leur démarche :

<https://github.com/robot-uprising-hq/ai-guide>

**8 - Conclusion.**

Le futur est en marche et jamais je n’aurai pensé être si proche de ce genre de chose et voir même peut être en être acteur, c’est en tout cas mon souhait pour l’avenir.

Il y’a 2 ans et demi je n’avais presque jamais écrit une ligne de code, entre temps j’ai découvert une passion, commercialisé un jeux en VR dans le monde, et me voilà près à créer des vrais robots intelligents, enfin c’est encore au stade du j’y pense en me rasant le matin (Aucun rapport avec le fait que je m'appelle Nicolas..).

Il est compliqué de tout évoquer, ce rapport aurait pu être beaucoup plus consistant et technique.

Je n’ai pas pu vous raconter par exemple le modèle qui avait complètement désobéi à mon code car il avait appris à terminer sa tâche plus rapidement en trichant.

Le jour où mon fils de 5 ans a mis plus de 15 minutes à enfin coincer la balle jaune du jeu.

Il faut bien comprendre qu’en restant que sur les acquis de ce projet, avec 2 scripts j’ai pu mettre en place 5/6 scènes de jeux, environ 1 jour par scène, et que rien que pour l’industrie du jeux vidéo, c’est un gain énorme en productivité pour les développeurs.

Unity a créé un service cloud de session de test de jeu avec le grid Sensor ou comment le Machine Learning vient détacher tout un pan du développement de produit par exemple.

Alors bien entendu, cela reste relativement complexe (mais de moins en moins), chronophage et peut être voué à l'échec si trop ambitieux, étape par étape.

Ce projet a été développé entre septembre 2020 et janvier 2021.

Autexier Nicolas.