BOUDON Alexandre
BRONGNIART Arthur
YE Steven

Projet
EzPath

Projet annuel 4 IABD

Sommaire

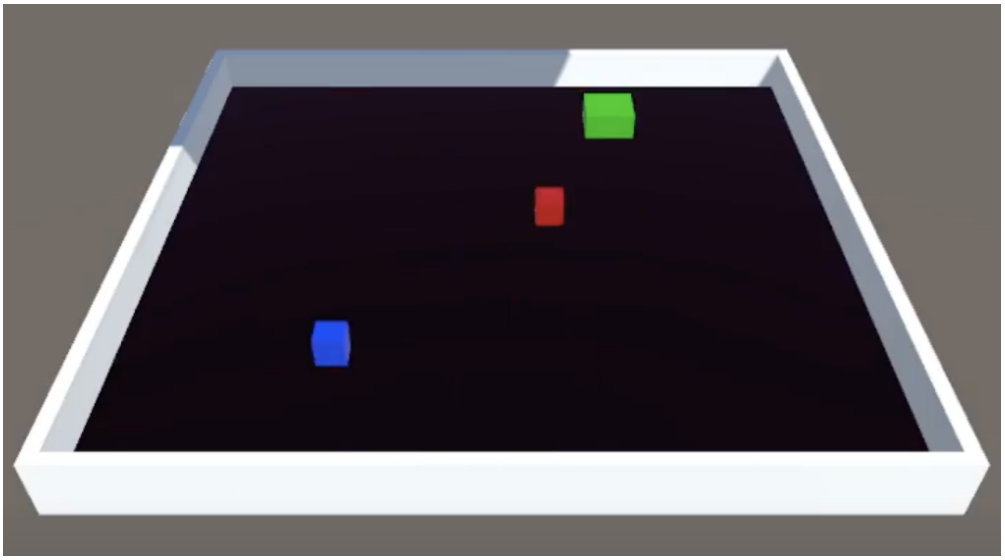
Introduction

L'utilisation de l'IA (Intelligence Artificielle) prend de plus en plus de place au sein des entreprises. Souvent sur le plan de l'optimisation et de la complexité des tâches, l'IA offre une possibilité intéressante par rapport à l'humain.
Les domaines d'applications sont larges mais soumis à de nombreuses contraintes telles que le coût, le temps, la difficulté et la connaissance nécessaire.

Une des contraintes qui nous avait été imposées était d'avoir un sujet basé sur le thème du transport, nous n'avions pas vraiment d'idée concrète au début mais en nous penchant sur les travaux du Hide&Seek d'OpenAI, nous voulions réaliser un projet qui pouvait y ressembler et s'appliquer dans un contexte entreprise. C'est de là qu'a émergé notre idée de créer un petit jeu où nous aurions une IA qui a pour objectif de transporter de la marchandise à un certain point.

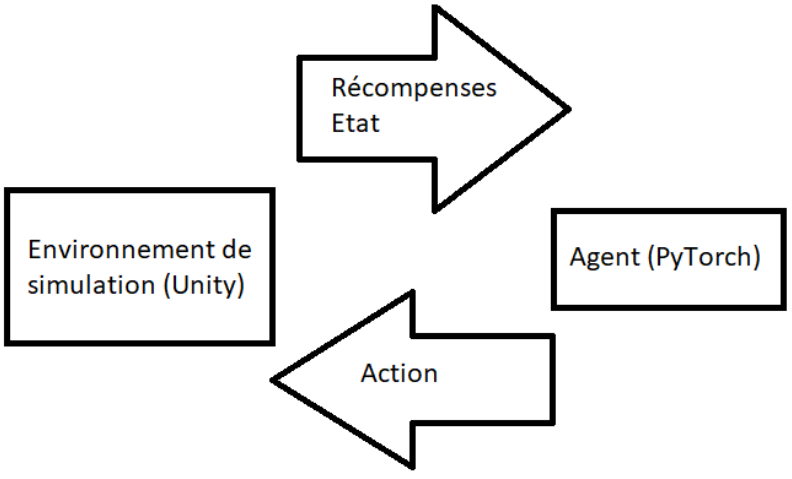
Présentation du projet

EzPath est un projet qui a pour but de créer une IA qui puisse aller d'un point A à un point B, rencontrant divers obstacles sur son chemin. Ce projet peut être utilisé à des fins logistiques dans le cadre du transport de biens dans un entrepôt ou dans une zone en guerre.
Notre objectif est de monter en complexité sur l'apprentissage de l'IA pour la généraliser sur plusieurs types d'environnements et de la rendre autonome, qu'elle soit seule ou en équipe.



Pour réaliser ce projet nous nous sommes penchés sur le logiciel de création d'environnement 3D Unity et du package python ML-Agents.

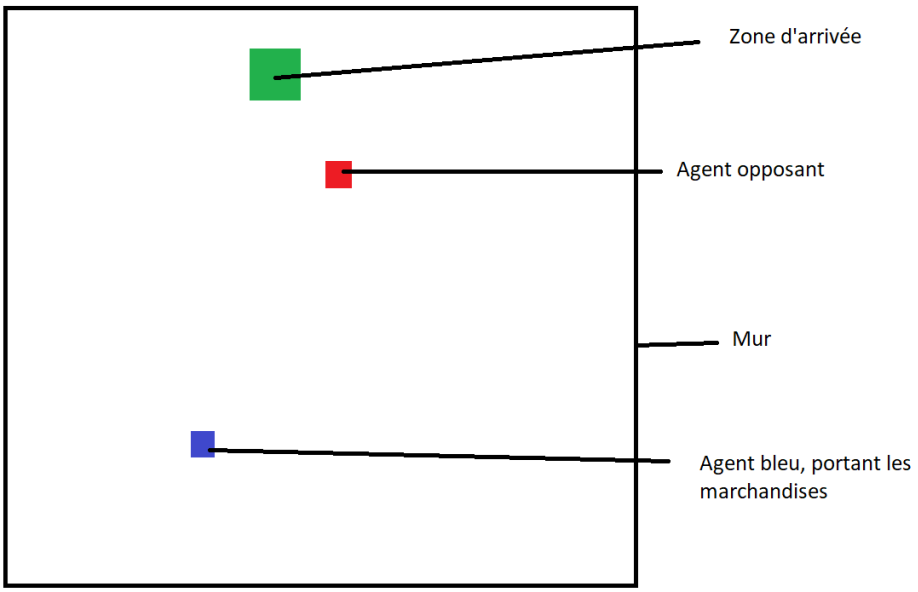
Unity est l'interface servant d'environnement de simulation pour l'apprentissage de notre IA tandis que ML-Agents s'occupe de l'interconnection entre Unity et le framework de machine learning: PyTorch. PyTorch recevra l'ensemble des informations de l'environnement pour prédire la meilleure action à effectuer et la renvoyer à Unity qui s'exécutera et donnera la récompense associée à celle-ci.



Résultat attendu

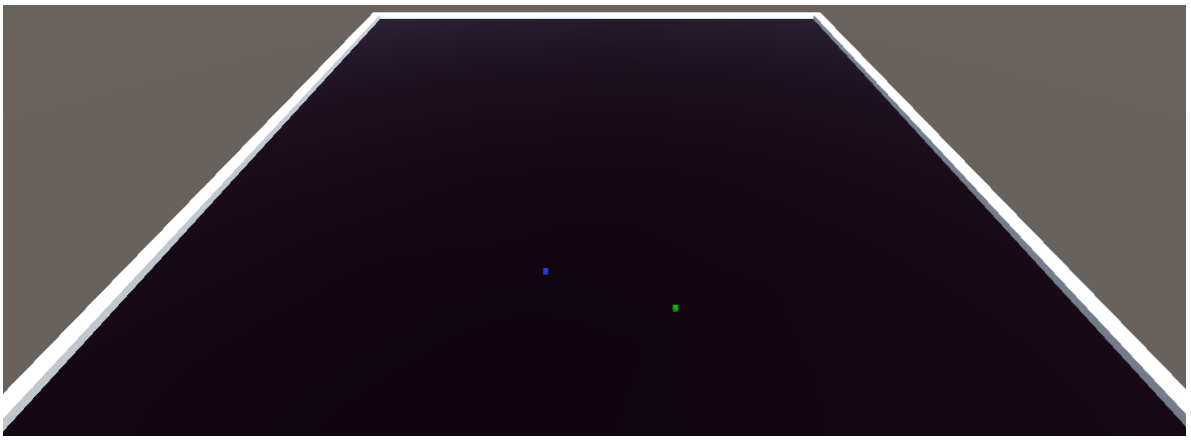
Afin de se donner un objectif à réaliser, il est essentiel de poser des objectifs à atteindre et pouvant répondre à de nombreux cas de logistique. Cette solution doit pouvoir s'adapter à diverses situations.

Notre IA, représentée par l'agent bleu ci-dessous, évolue dans un environnement en ayant un champ de vision qui peut être défini sous la forme de capteurs autour d'elle. Son objectif est d'atteindre une zone d'arrivée tout en évitant les murs et l'agent opposant qui se dresseront devant elle dans une limite de temps donnée. À préciser que notre agent a accès au vecteur de distance entre elle et la zone (différence entre ses coordonnées et les coordonnées de la zone verte).



A partir d'un apprentissage sur un environnement basique, notre agent bleu rencontrera diverses modifications dans son environnement, l'obligeant à s'adapter pour atteindre la zone de victoire :

Un environnement où l'agent bleu doit apprendre à se diriger vers la zone de victoire dans une zone assez large.



Cet environnement sert à apprendre à l'agent bleu à se diriger vers la zone de victoire. L'agent bleu et la zone de victoire apparaissent dans une zone très grande de manière aléatoire.
Le temps pour chaque épisode est suffisamment long pour traverser la zone en ligne droite. Il faut donc que l'agent bleu apprenne à se diriger vers la zone de victoire de façon relativement optimisée pour obtenir sa victoire.

Cet environnement est une vision simpliste de nos autres environnements qui permet d'apprendre les règles plus basiques de notre jeu : ne pas toucher les murs et aller vers la zone de victoire de manière optimale.

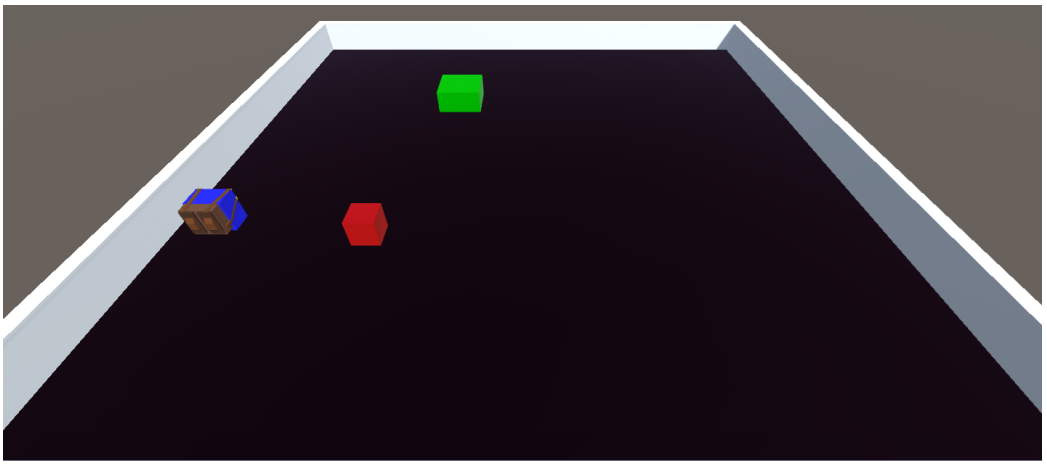
Dans cet environnement, notre agent bleu arrive assez rapidement à apprendre les règles. Sans faire le trajet le plus optimisé, au bout de quelques dizaines de milliers d'épisodes, notre agent atteint presque tout le temps la zone de victoire.

Parfois, on obtient des comportements singuliers sur certains épisodes:

- L'agent bleu tourne autour de la zone de victoire sans la toucher
- L'agent bleu oscille un peu pour traverser le terrain

Ces comportements peuvent être expliqués par la curiosité implémentée dans son apprentissage mais d'autres facteurs doivent aussi entrer en jeu.

Un environnement où l’agent bleu apprend à esquiver l’agent rouge en plus des contraintes précédentes.



Cet environnement est très proche du précédent, à l'exception que le terrain est plus petit et qu’il y a un agent rouge. Cet agent rouge est utile pour notre cas, en effet il peut être considéré comme un facteur inconnu avec un comportement aléatoire voire hostile envers notre agent bleu (par exemple un humain, un drone ou un quelconque obstacle imprévu).
Le temps pour traverser le terrain est beaucoup plus court que précédemment.

Là aussi, une vision plus simpliste de la réalité est représentée, les seules règles sont de toucher la zone de victoire, de ne pas toucher les murs et éviter l’agent rouge dans le temps imparti.
L’agent rouge est aussi une IA qui doit apprendre à toucher l’agent bleu : il reçoit une récompense s’il touche l’agent bleu ou si le temps imparti est écoulé.

L’apprentissage de notre agent bleu est plus long que sur le précédent. Comme les deux agents apprennent au fur et à mesure que le nombre d'épisodes augmente, la difficulté pour l’un et l’autre augmente aussi.

Malgré tout, l'agent bleu arrive à apprendre à faire des feintes et esquiver l’agent rouge. L’agent rouge quant à lui arrive de temps à temps à contrer les feintes (il apprend à se diriger vers l’agent bleu et parvient à “prédire” certaines feintes).

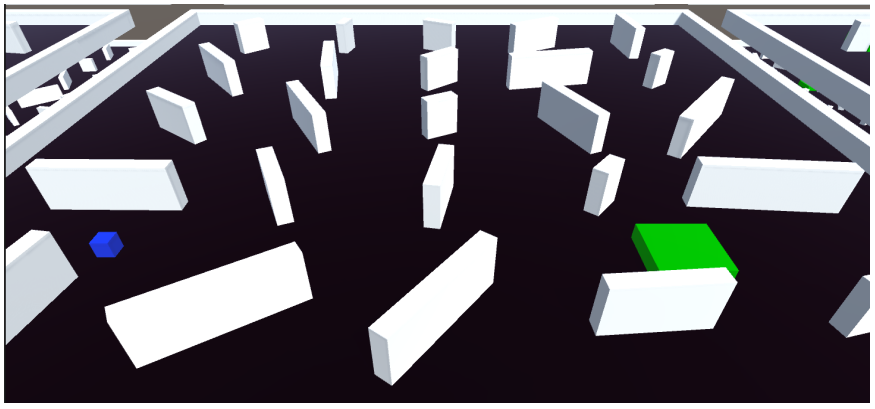
Nous avons vu plusieurs comportements notables lors des différents entraînements :

- Les agents se rapprochent des murs en allant dans une unique direction.
- L'agent bleu se faisait pousser dans un coin par l'agent rouge, n'arrivait plus à attendre la zone de victoire et se déplaçait légèrement de droite à gauche. L'agent rouge, une fois à côté du bleu, n'osait pas toucher l'agent bleu.

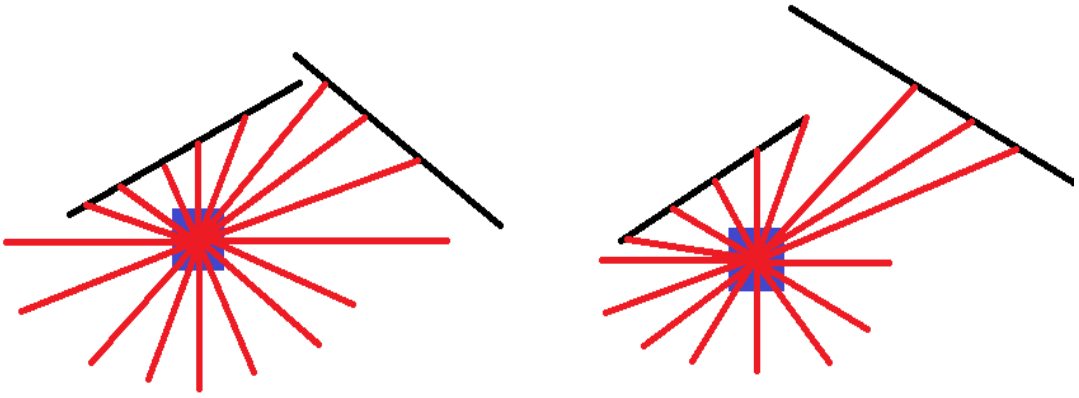
Pour ces comportements négatifs ci-dessus, nous avons changé les hyperparamètres ainsi que les récompenses pour forcer l'agent bleu à être plus agressif.

- L'agent bleu a tendance à favoriser un côté plutôt qu'un autre quand il fait ses feintes. Cela dit, il change de côté une fois que l'agent rouge a appris de quel côté est la feinte.
  On peut noter que ce comportement est aussi visible chez les humains.

Un environnement avec des murs de taille et position aléatoires que notre agent bleu n'aura pas le droit de toucher.



Cet environnement se rapproche d'une simulation plus concrète et plus utile dans un cas réel. Il est même plus complexe que la réalité par rapport à un entrepôt. Et de son point de vue, il peut être difficile de percevoir les espaces entre les murs.



Difficulté rencontrée vis-à-vis de l'utilisation du Raycasting

Les murs ne sont pas placés aléatoirement car cela pourrait amener à avoir des épisodes sans solution (les murs enferment la zone verte), ce qui correspondrait dans la réalité à des missions qui ne peuvent qu'échouer. Pour rendre aléatoire notre environnement, nous avons opté pour un agrandissement en longueur aléatoire et une rotation aléatoire en vérifiant qu'il y est toujours un passage possible.

On espérait pouvoir permettre à l'agent bleu de généraliser et de trouver la zone de victoire quelque soit le terrain. Sa position de départ et la position de la zone de victoire sont aussi aléatoires.
On voulait qu'il apprenne à chercher un chemin jusqu'à la zone de victoire en esquivant l'agent rouge et les murs.

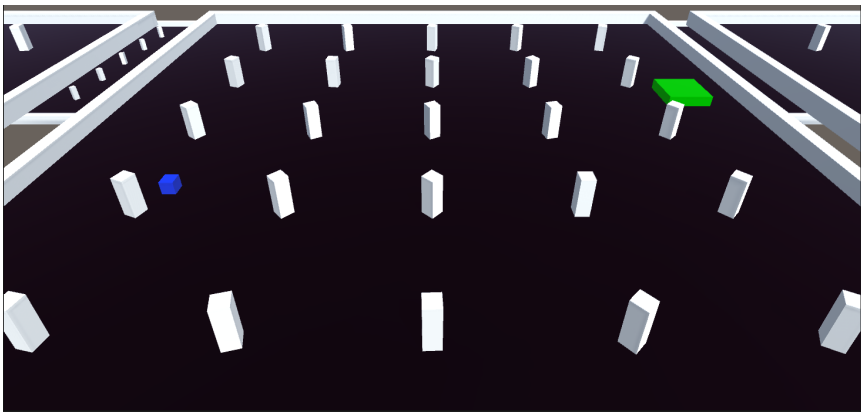
Malheureusement, notre agent bleu n'a pas été capable d'apprendre, sûrement dû à la trop grande complexité qu'offre l'environnement.

Nous avons essayé les trois algorithmes d'apprentissage que propose ML-Agent : POCA, SAC et PPO. Le SAC ne fonctionne plus très bien depuis la migration de ML-Agent sur pyTorch (précédemment Tensor Flow) et même s'il semblait fournir les meilleurs résultats, il augmentait notre temps d'apprentissage par dix (par rapport à PPO et POCA), ce qui n'était pas gérable.

Nous avons essayé de changer les hyperparamètres : la curiosité, le learning rate, la taille du modèle mais cela n'a pas fonctionné. Nous avons même changé les récompenses et désactivé la défaite si un mur était touché pour simplifier l'apprentissage.

Nous pensions que le problème venait peut-être du fait que l'environnement était trop complexe pour notre IA, par conséquent nous avons réduit nos murs à de simples piliers fixes et nous avons centré la zone de victoire. Après quelques heures, l'IA avait réussi à apprendre et parvenait à générer des résultats positifs.

Au vu de ces résultats, nous avons décidé de rendre la position de la zone verte aléatoire, cependant après 24 heures d'entraînement notre petit agent bleu avait complètement oublié son apprentissage précédent et ne bougeait plus du tout. Nous avons essayé de changer les récompenses pour qu'il reçoive un malus très négatif s'il restait immobile, un peu moins négatif s'il prenait un mur et ce dans le but de le faire bouger mais cette modification n'a pas fonctionné.



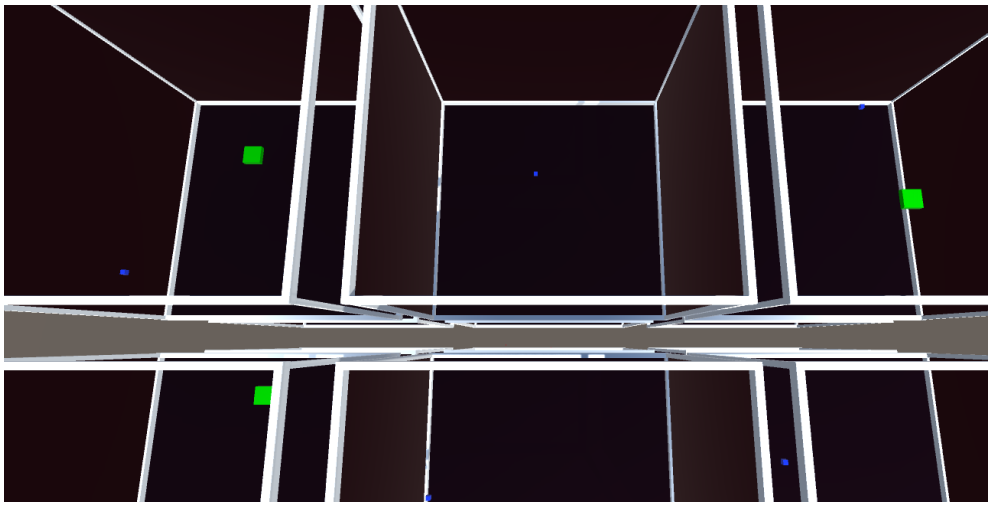
Nous avons donc choisi d'ajouter de nouveau sensor pour l'aider à se repérer dans l'espace et nous avons ajouté l'ensemble des coordonnées des murs. Le défaut principal de cette méthode est qu'il faut connaître à l'avance l'ensemble des murs composant l'environnement, dans un entrepôt ce n'est pas problématique mais pour un déplacement extérieur cela n'est pas possible.

Malgré tout nous avions espoir qu'avec cette méthode l'agent bleu puisse se déplacer plus facilement entre les murs et trouver un chemin plus optimal. Cela permet aussi de compenser la plupart des défauts du raycasting le tout pour un relatif faible coût de calcul supplémentaire.
Cette méthode s'est aussi montrée infructueuse surement à cause de nos récompenses, hyperparamètre et/ou des algorithmes d'apprentissage proposés par ML-agent. Nous n'avons donc pas réussi à adapter notre IA pour n'importe quel type d'environnement.

Après réflexion, nous avons trouvé quelques solutions qui pourraient résoudre notre problème :

- Enlever l'aspect aléatoire de la zone de victoire
- Enlever l'aspect aléatoire de la position de notre agent bleu et de nos murs, cela permettra peut-être à notre agent d'apprendre une "carte" sur laquelle se déplacer sans générer de collisions (Non testé)
- Changer le mode vision pour un mode caméra qui permettra une meilleure vision et reconnaissance de l'environnement (Non testé). Cette solution demande des ressources trop grandes en temps et/ou en coût pour nous.

Un environnement 3D, où les agents se déplacent aussi verticalement, avec des obstacles en hauteur



Cet environnement simule un espace 3D. Notre agent bleu peut se déplacer librement en hauteur, largeur et longueur. Il représente un monde dans lequel un drone serait, mais en plus restreint. Si on souhaite l'adapter à la réalité, il faudrait définir le plafond comme étant la hauteur maximale que peut atteindre le drone (qui peut être variable en fonction des conditions météorologiques par exemple), les murs quant à eux seraient des réels obstacles comme des bâtiments, des camions, etc…
Les règles du jeu sont donc très simples : ne pas toucher les murs et atteindre la zone de victoire dans le temps imparti.

Nous n'avons pas pris le temps de beaucoup travailler dans cet environnement et il devait se complexifier par la suite une fois un résultat d'apprentissage correct obtenu. Notamment par l'ajout d'objet volant fixe (pouvant représenter des murs, des

branches, etc…) et des agents rouges (pouvant représenter des oiseaux ou d'autres drônes).

Avec nos entraînements, on peut noter qu'il a du mal à toucher la zone verte et à se stabiliser (l'agent bleu est soumis à la gravité). Cela s'explique très facilement par le fait que les raycasting sont en 2D et ne permettent pas d'observer au-dessus ou en dessous de l'agent. Malgré tout, il devrait pouvoir se repérer avec le vecteur distance entre lui et de la zone de victoire.

Pour améliorer l'apprentissage il serait intéressant d'ajouter des sensors de type raycast sur les différents plan pour lui permettre un meilleur scan.

Un environnement avec plusieurs agents bleus (avec des récompenses différentes sur chaque agent) et rouges, permettant d'avoir une notion d'équipe

Cet environnement a été pensé à la création du projet dans le cas où nous obtiendrions des résultats concluants sur nos précédents environnements. Nous voulions entraîner les agents bleus à interagir en équipe.

On souhaitait attribuer des valeurs de récompense différentes sur chaque agent bleu et déterminer l'agent avec le plus de valeurs pour le "protéger". Mais cela demande l'utilisation du SAC et POCA car ce sont des algorithmes de groupe et qui peut apprendre aussi par l'expérience des personnes de son groupe.

Un environnement où les agents ont accès à plus d'actions et peuvent lancer des compétences spéciales

> Cet environnement était aussi en bonus et se rapproche très fortement de l'idée du jeu Hide&Seek d'OpenAI. Une fois que nos agents (bleu et rouge) ont bien appris les règles, nous voulions observer quel comportement ils allaient avoir si on leur ajoutait des "pouvoirs" qu'ils pouvaient activer à certains moments.
>
> Notamment observer s'ils allaient comprendre comment les utiliser de manière optimale : avec un pouvoir de courir plus vite pendant 1 seconde qui est activé pour esquiver ou toucher un agent, un pouvoir ralentissant les personnes autour, pouvoir passer à travers un mur ou sauter par-dessus…
>
> Nos idées ne manquaient pas mais le temps oui. Même si nous ne possédons pas les mêmes ressources qu'OpenAI.

À travers ces divers apprentissages dans des environnements différents les uns des autres, l'objectif attendu est que l'agent bleu puisse s'adapter à plusieurs cas d'usage.

Cependant, l'évolution de l'agent bleu ne s'est pas faite aussi facilement. De nombreuses problématiques ont été soulevées, amenant à des difficultés sur divers points comme l'apprentissage ou l'environnement en lui-même.

Problèmes rencontrés et solutions trouvées

Lors de l'évolution des environnements, nous nous sommes heurtés à plusieurs problématiques où nous avons dû trouver des alternatives ou changer notre ligne directrice.

- L'algorithme d'apprentissage non optimisé pour notre problème
  Nous avons essayé les trois algorithmes proposés par ML-agents : PPO, POCA et SAC:
  - PPO est un algorithme puissant qui permet d'entraîner un unique agent, il est puissant dans l'estimation d'une action par rapport à une action moyenne pour un état mais peut générer du bruit lors de la mise à jour de la stratégie (policy).

  - POCA est un algorithme qui permet d'entraîner un groupe d'agents. Le réseau de neurones va gérer l'ensemble de l'équipe et donne des récompenses de groupe mais aussi personnels pour apprendre à mieux contribuer en équipe pour obtenir les récompenses, ce qui n'est pas recherché pour la plupart de nos environnements.

  - SAC est le dernier algorithme proposé par ML-Agents, il a reçu une mise à jour (passage de Tensor Flow à PyTorch) et pose un problème d'apprentissage sur nos machines (notre environnement gèle pendant plusieurs minutes) ce qui augmente le temps d'apprentissage par dix. Nous n'avons donc pas pu vraiment travailler avec.

- Un apprentissage trop long
  Lors des derniers apprentissages sur des environnements un peu plus poussés, nous nous sommes heurtés aux limites des possibilités d'apprentissage de notre agent.

- La répartition des récompenses
  Les récompenses positives ou négatives influent considérablement sur les choix que va effectuer notre IA. De plus, un biais lié à notre perception de la meilleure solution rentre en jeu.

- La vision de l'agent

  La vision de l’agent est déterminée par des lasers (Raycasting) qui lui permettent d’estimer les distances. Mais ces lasers ne permettent pas d’avoir une vision totale (d’un point de vue aérien) et peut rater des ouvertures et/ou des informations sur l’agent rouge et la zone de victoire.

- La gestion AWS

  Avec le planning fixé en début de projet, nous comptions sur notre expérience d'AWS pour déployer rapidement notre projet sur un EC2 grâce à l'utilisation de terraform. En automatisant la création de l'EC2, nous espérions pouvoir maîtriser plus facilement le coût des machines.

  Cependant, à cause d'une erreur de paire de clé postée en public sur github, le compte a été bloqué et nous n'avons pu faire aucune manipulation par la suite. Il aura fallu attendre plusieurs jours afin que le compte soit débloqué et pour que nous puissions recommencer l'automatisation du déploiement des EC2.

Malgré ces nombreux problèmes rencontrés, quelques solutions ont été trouvées afin de pouvoir continuer notre apprentissage correctement. Cela n'est cependant pas le cas pour tous les problèmes, nous obligeant dans certains cas à faire marche arrière.

Afin de pallier aux différentes problématiques rencontrées, nous avons trouvé diverses solutions nous permettant de ne pas revenir sur nos pas dans la plupart des cas que nous avons pu avoir.

Voici une liste de différentes solutions suite aux problèmes rencontrés:

- La solution AWS
  Afin de pouvoir utiliser à nouveau AWS, nous n'avons eu d'autres choix que d'utiliser un autre compte que celui utilisé précédemment. Le compte principal étant toujours bloqué suite aux vérifications de la part du support d'AWS, nous avons dû migrer notre installation.

  Par chance, passant par terraform afin de faciliter l'installation, la configuration du nouveau compte a été plus rapide que prévu et nous a permis de gagner du temps pour les différentes tâches en cours en parallèle.

- Les apprentissages trop longs
  La complexité de l'apprentissage a grandement ralenti notre progression pour le projet. Nous avons constaté que partir d'un nouvel apprentissage sur la version finale d'un environnement ne donnerait aucun résultat satisfaisant sur un délai aussi court.

  Afin de maximiser les chances d'apprentissage et de réussite de notre agent, nous avons décidé de décomposer nos différents environnements et de réutiliser le même apprentissage à chaque ajout d'un composant de l'environnement.

  L'objectif était que l'agent puisse apprendre dans un environnement évoluant au fur et à mesure, en se basant sur un apprentissage d'un environnement similaire mais plus simple pour celui-ci

Explication du code

Afin de comprendre le fonctionnement de nos différents agents, nous allons étudier en détail les hyper-paramètres choisis. Nous nous consacrerons essentiellement à notre agent bleu. Nous avons décidé de ne pas détailler les hyper-paramètres de l'agent rouge (catcher) car nous avons jugé cela peut pertinent.

```
behaviors:
  Catcher:
    [...]
  DeliveryMan:
    trainer_type: ppo ← Choix de l'algorithme DRL le plus adapté
    hyperparameters:
      batch_size: 128 ← Nombre d'épisodes pris en compte pour faire une
évaluation du modèle
      buffer_size: 2048 ← Nombre d'épisodes nécessaires pour faire une
mise à jour des poids du modèle
      learning_rate: 3.0e-4 ← Le coefficient de modification des poids
du modèle après chaque apprentissage
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings:
      normalize: false ← permet de lisser les valeurs dans l'intervalle
[0; 1]
      hidden_units: 256 ← nombre de neurones par couche du modèle
      num_layers: 12 ← nombre de couches cachées du modèle
    reward_signals:
      extrinsic: ← Définit les signaux de récompenses externes
        gamma: 0.99 ← Distance à laquelle l'agent va se soucier des
récompenses à obtenir
        strength: 1.0 ← Facteur multipliant les récompenses donné par
l'environnement
      curiosity: ← Permet de définir curiosité de l'agent
            gamma: 0.9
            strength: 0.01
            learning_rate: 0.001 ← Le coefficient de modification des
poids du modèle après chaque apprentissage (si la curiosité est
impliquée)
    keep_checkpoints: 150 ← Nombre maximum de checkpoints à enregistrer
pour le modèle (suppression du plus vieux une fois atteint)
    max_steps: 3000000000 ← Nombre d'étapes avant la fin d'un épisode
```

```
    time_horizon: 1000
    summary_freq: 10000
    self_play:
      save_steps: 50000
      team_change: 200000
      swap_steps: 4000
      window: 10
      play_against_latest_model_ratio: 0.5
      initial_elo: 1200.0
```

Nous passons ensuite sur le contenu du code permettant la création de l'environnement, la gestion des récompenses à attribuer, la réinitialisation d'une scène et les déplacements des agents.

```
public void ResetScene()
    {
        resetTimer = 0;
        numberBlueAgentAlive = numberBlueAgent;
        ResetInsideWall();

        //Reset Agents
        foreach (var item in AgentsList)
        {
            bool haveColision = true;
            var newStartPos = new Vector3(0f, 0f, 0f);

            while (haveColision)
            {
                newStartPos = item.Agent.initialPos + setRandomPosition();
                if (!Physics.CheckSphere(newStartPos, radiusSpawnCube + 0.5f))
                {
                    haveColision = false;
                }
            }

            var newRot = Quaternion.Euler(0, 0, 0);
            item.Agent.transform.SetPositionAndRotation(newStartPos, newRot);

            item.Rb.velocity = Vector3.zero;
            item.Rb.angularVelocity = Vector3.zero;
        }

        ResetVictoryZone();
    }
```

Le code ci-dessus est la fonction qui est appelée à chaque nouvel épisode. Il réinitialise l'environnement comprenant la remise à zéro du temps, le nombre d'agents bleu (dans le cas d'un environnement avec de multiples agents bleus) et la régénération des dimensions et rotations des murs

Ensuite, on itère sur l'ensemble de nos agents (bleus et rouges). Pour chaque agent parcouru, nous leur définissons :

- Une nouvelle position de départ avec une vérification de non contact avec un quelconque objet (mur, autre agent). Dans le cas d'un contact, une nouvelle position sera générée jusqu'à ce qu'il n'y ait plus de contact.
- Réinitialisation de la vitesse de l'agent afin qu'il ne garde pas l'inertie de l'épisode précédent

Une fois tout cela créé, on instancie aussi la zone de victoire.

```
public void MoveAgent(ActionSegment<int> act)
    {
        var dirToGo = Vector3.zero;

        var forwardAxis = act[0];
        var lateralAxis = act[1];

        dirToGo += moveForwardAxis(forwardAxis);
        dirToGo += moveLateralAxis(lateralAxis);

        if (dirToGo.z == 0f || dirToGo.x == 0f) //pas de mouvement en diagonal
        {
            agentRb.AddForce(dirToGo, ForceMode.VelocityChange);
        }
        else
        {
            agentRb.AddForce(dirToGo * 0.71f, ForceMode.VelocityChange);
        }
    }
```

Notre fonction MoveAgent permet de gérer les déplacements de nos agents, on y récupère un actuator qui contient deux valeurs allant de 0 à 2. Ces valeurs permettent de savoir dans quelle direction on veut aller et sont envoyées dans les fonctions moveForwardAxis et moveLateralAxis qui les convertiront en Vector3, traduisant les actions en déplacement.
Aussi, il est possible d'envoyer un déplacement latéral et longitudinal en même temps, et si c'est le cas, on multiplie le vecteur de déplacement par 0.71 permettant d'égaliser la distance parcourue avec le cas où on n'a qu'une seule action.

```
public void PointScored(int result){
        if (numberBlueAgentAlive == 1)
        {
            if (result == 0) //Blue win
            {
                foreach (var item in AgentsList)
                {
                    if (item.Agent.team == Team.Blue)
                    {
                        item.Agent.giveRewardBlue(rewardValue * 5, resetTimer,
MaxEnvironmentSteps);
                    }
                }

                RedAgentGroup.AddGroupReward(-(rewardValue - (float) resetTimer
/ MaxEnvironmentSteps));
            }
            else if (result == 1) //red win
            {
                RedAgentGroup.AddGroupReward(rewardValue); //-
(float)m_ResetTimer / MaxEnvironmentSteps
                foreach (var item in AgentsList)
                {
                    if (item.Agent.team == Team.Blue)
                    {
                        item.Agent.malusRedTouchBlue(rewardValue * 3);
                    }
                }
            }
            else if (result == 2) // blue touch wall
            {
                foreach (var item in AgentsList)
                {
                    if (item.Agent.team == Team.Blue)
                    {
                        item.Agent.malusWallTouch(rewardValue, resetTimer,
MaxEnvironmentSteps);
                    }
                }
            }
            RedAgentGroup.EndGroupEpisode();
            BlueAgentGroup.EndGroupEpisode();
            ResetScene();
        }
        else
        {
            numberBlueAgentAlive -= 1;
        }
    }
```

La fonction PointScored est appelée à chaque fois qu'un agent doit gagner ou perdre des points, attribuant les récompenses à son équipe. Si le dernier agent bleu en vie marque des points, la fonction permettant de réinitialiser l'environnement est appelée et un nouvel épisode est lancé.

La fonction est construite de la manière suivante :

- On commence par vérifier si l'agent bleu est le dernier vivant
    - Si oui :
        - Si les agents bleus ont gagné :
          Chaque agent bleu reçoit son bonus respectif (ce qu'il obtient seul)
          Les agents rouges reçoivent leur malus respectif.

        - Sinon si les rouges ont gagné:
          Les agents bleus reçoivent leur malus respectif.
          Les agents rouges reçoivent leur bonus respectif.

        - Sinon si les bleus touchent un mur :
          Les agents bleus reçoivent le plus grand malus.

      Les agents reçoivent leurs récompenses gagnées lors l'épisode (en tant que groupe), l'épisode s'arrête et l'environnement est réinitialisé

    - Sinon :
      On réduit le nombre d'agent bleu de un et l'épisode continue

Conclusion

Un **projet inachevé par manque de temps et de matériel.**

Comme évoqué ci-dessus, les objectifs finaux de ce projet demandaient beaucoup de **temps et de ressources** que nous n'avions pas à disposition d'un point de vue:

- Matériel, nous ne sommes en possession que d'un seul ordinateur capable de prendre en charge le projet sur un si long temps d'apprentissage,
- Temporel, les apprentissages peuvent prendre entre 48 heures et 2 semaines, réduisant les possibilités de varier les apprentissages en attendant les premiers résultats.

Malgré le faible taux de résultats positifs sur les derniers environnements mis en place, ce projet nous a permis d'en apprendre davantage sur l'utilisation de Unity et de sa librairie ML-Agents. Travailler sur un projet d'intelligence artificielle avec un réel visuel était notre volonté depuis le début de ce projet et **nous ouvre de nouvelles opportunités pour l'année prochaine**.

Une nouvelle approche du cloud et de l'intelligence artificielle:

Avec l'approfondissement du service Cloud que propose AWS, cela nous a donné la chance d'expérimenter l'apprentissage de l'IA avec le cloud. Grâce à cette ouverture sur le Cloud, nous avons vu la puissance de cette technologie mais aussi ses contraintes pour un projet (coût, compétences).

Nous comptons à l'avenir, utiliser davantage cette connexion entre le cloud et l'intelligence artificielle (sous réserve de fonds) afin de combler les différents problèmes rencontrés lors de ce projet annuel. De même pour Unity et ML-Agents que nous continuerons à utiliser si possible lors de nos prochains projets.

Ce projet a été enrichissant sur tous les points que nous avons pu aborder, même si nous n'avons pas pu atteindre tous les objectifs souhaités. Lorsque la date limite de rendu sera passée, nous souhaitons continuer à chercher des solutions aux objectifs que nous n'avons pas pu atteindre, et ce dans le but de nous améliorer et d'aller de l'avant.