Rapport de projet

Document officiel / PROROK

Apprentissage automatique de la marche pour un robot quadripède

DULOUARD Clément
DEME Rémy

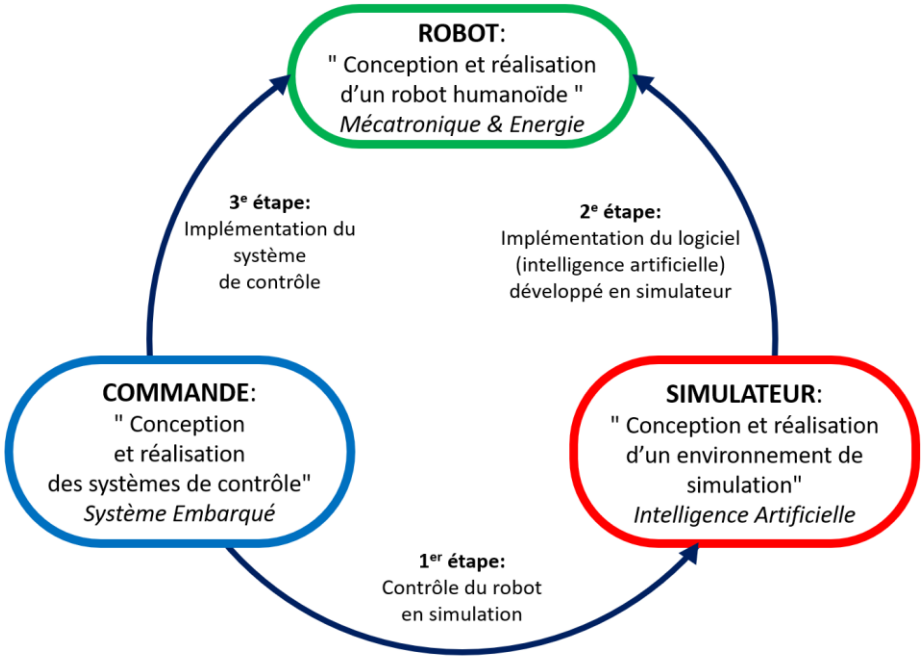
PROROK

Sommaire :

Introduction :

Ce projet s'inscrit dans le cadre du projet inter-majeure PROROK. L'association PROROK a pour but de développer un projet de robot humanoïde de grande taille, piloté partiellement par intelligence artificielle .
Afin de mener ce projet à bien, il est primordial de maîtriser un grand nombre de technologie. De ce fait, nous avons donc commencé cette année par concevoir un prototype de robot humanoïde de 1 m.
Pour ce faire, nous avons choisi de développer nos compétences sur différents projets comme le contrôle de moteurs, la récupération de données auprès d'un opérateur humain, et, bien sûr, ce projet de simulateur afin de pouvoir commencer à manipuler des algorithmes d'apprentissage automatique.

Le modèle de robot sur lequel nous avons choisi de travailler est un modèle de robot quadripède à douze articulations inspirées du robot SpotMini de Boston Dynamics.



Motivations :

Les raisons qui nous ont poussés à choisir ce projet sont multiples.
Premièrement, ce projet est le premier projet de grosse envergure que l'association a à gérer. Cela nous permet donc de tester nos compétences en matière de gestion et de développement de projet.
Deuxièmement, ce projet nous permet de découvrir les problématiques de la simulation et de l'apprentissage automatique. Grâce aux retours de ce projet, nous allons pouvoir développer un nouveau simulateur pour entrainer le robot humanoïde de façon plus efficace tout en évitant les problèmes que nous avons eus à résoudre sur ce simulateur. De plus, ce simulateur nous permet de tester divers algorithmes et méthodes d'apprentissage de façon simple.
Enfin, mener ce projet à bien est pour nous, et, pour l'association, une réelle fierté et une preuve de notre savoir-faire, de notre sérieux et de notre efficacité dont PROROK souhaite faire sa marque de fabrique.

Nous vous souhaitons une bonne lecture et espérons que vous apprécierez le travail que nous avons fourni.

Objectifs :

Les objectifs que nous nous sommes fixés pour ce projet sont les suivants :

- Acquérir des compétences de développement sur le moteur de jeux Unity
- Proposer une simulation d'un robot proche de la réalité
- Proposer une simulation polyvalente et facilement adaptable
- Proposer un système de communication simple pour permettre de tester divers modèles de pilotage du robot
- Acquérir des connaissances concernant l'apprentissage automatique notamment en robotique
- Proposer une solution d'apprentissage automatique afin de faire marcher le robot dans un environnement « simple » (Avancer tout droit sur un sol plat et sans obstacles)
- Mettre en application notre modèle d'apprentissage sur notre simulateur

Méthodologie :

Cette partie se divise en trois sous-parties :
Tout d'abord, une rapide présentation du simulateur afin de permettre d'offrir une vision globale des fonctionnalités disponibles.
Ensuite, une explication du fonctionnement du simulateur dans laquelle nous étudierons les différentes fonctions importantes du simulateur et où nous expliquerons les différents choix que nous avons dû faire et les problèmes que nous avons pu rencontrer.
Enfin une dernière partie détaillera le travail de recherche qui a été fait concernant l'apprentissage automatique et les solutions que nous avons développées pour répondre à nos problématiques.

Important :

Il n'est pas nécessaire d'ouvrir le projet dans l'éditeur Unity ni d'ouvrir le code en complément du rapport pour comprendre de quoi il est question car les morceaux de codes ou les fonctionnalités dont nous parlons sont illustrés par des captures d'écran.

Cependant si celles-ci ne suffisent pas, le code du simulateur est disponible sur un repository GitHub à l'adresse suivante :

https://github.com/CDulouard/ProrokTestUnit2

Le projet Unity se trouve dans le dossier ProrokUnitTest2V3.
La version d'Unity utilisée pour le développement est Unity 2018.3.11f1, néanmoins les autres versions d'Unity 2018 devraient être en mesure d'ouvrir le projet.
Pour accéder directement aux scripts sans ouvrir le projet avec Unity, vous pouvez accéder au dossier Scripts contenu dans le dossier Assets (Les scripts de l'UI sont dans un sous dossier des Scripts nommé UIScripts).

Si vous souhaitez essayer le simulateur compilé, vous trouverez sur GitHub à la racine du repository, trois dossiers compressés (Linux.rar, Windows.rar, MacOS.app.rar) qui contiennent le simulateur compilé pour les trois principaux OS. Cependant seule la version Windows a été testée, nous ne garantissons donc pas le fonctionnement des deux autres versions (bien qu'il soit peu probable qu'il y ait un dysfonctionnement).
Le simulateur a été conçu pour fonctionner à une résolution de 1920 * 1080. L'UI a été paramétrée pour s'adapter à différentes résolutions mais il est possible qu'il y ait des problèmes d'affichages mineurs si la résolution est changée.

Des connaissances de base en c# et sur Unity peuvent être nécessaires pour la bonne compréhension du projet. Vous trouverez des cours afin d'avoir ces connaissances aux adresses suivantes :
https://unity3d.com/fr/learn/tutorials
https://openclassrooms.com/fr/courses/1526901-apprenez-a-developper-en-c

Présentation du simulateur :

Cette section a pour but de présenter les fonctionnalités proposées par le simulateur. Nous rentrerons dans les détails de son fonctionnement dans la partie intitulée Fonctionnement du simulateur. La lecture de cette partie n'est pas nécessaire à la compréhension du projet, elle permet néanmoins d'avoir un aperçu rapide de ce qu'offre le simulateur et explique comment l'utiliser.

Menu principal

Figure 1

Le menu principal est le point d'entrée du simulateur, il propose trois possibilités :

- Accéder au mode de simulation qui permet de visualiser le robot ainsi que ses outils de supervision
- Accéder au mode d'entrainement qui permet de ne voir que les outils de supervision du robot afin de diminuer la consommation de ressources lors de l'entrainement du robot
- Accéder aux paramètres de la simulation qui permet d'ajuster celle-ci à nos besoins sans devoir modifier le code ni les paramètres d'Unity

Paramètres

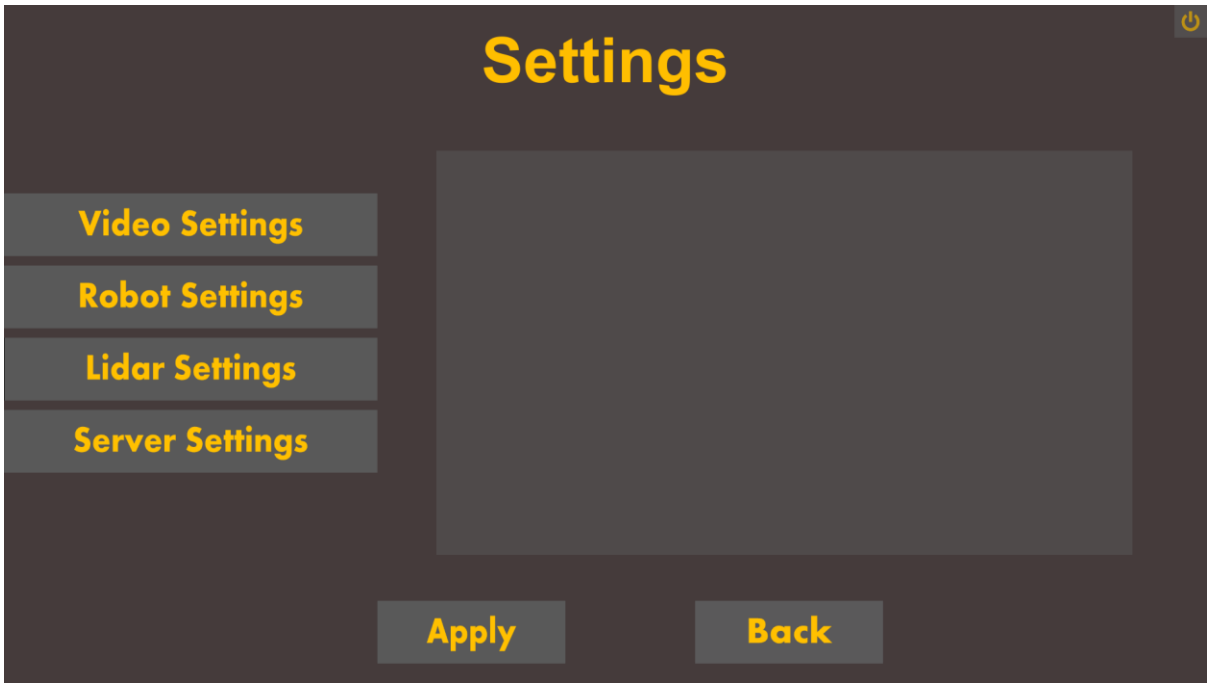


Figure 2

Les paramètres permettent de modifier quatre types de réglages : les réglages vidéo, les réglages du robot, les réglages du lidar et les réglages du serveur. Une fois les paramètres souhaités entrés, il faut appuyer sur le bouton « Apply » pour les valider et les sauvegarder dans le fichier de paramétrage du simulateur.

Paramètres vidéo :

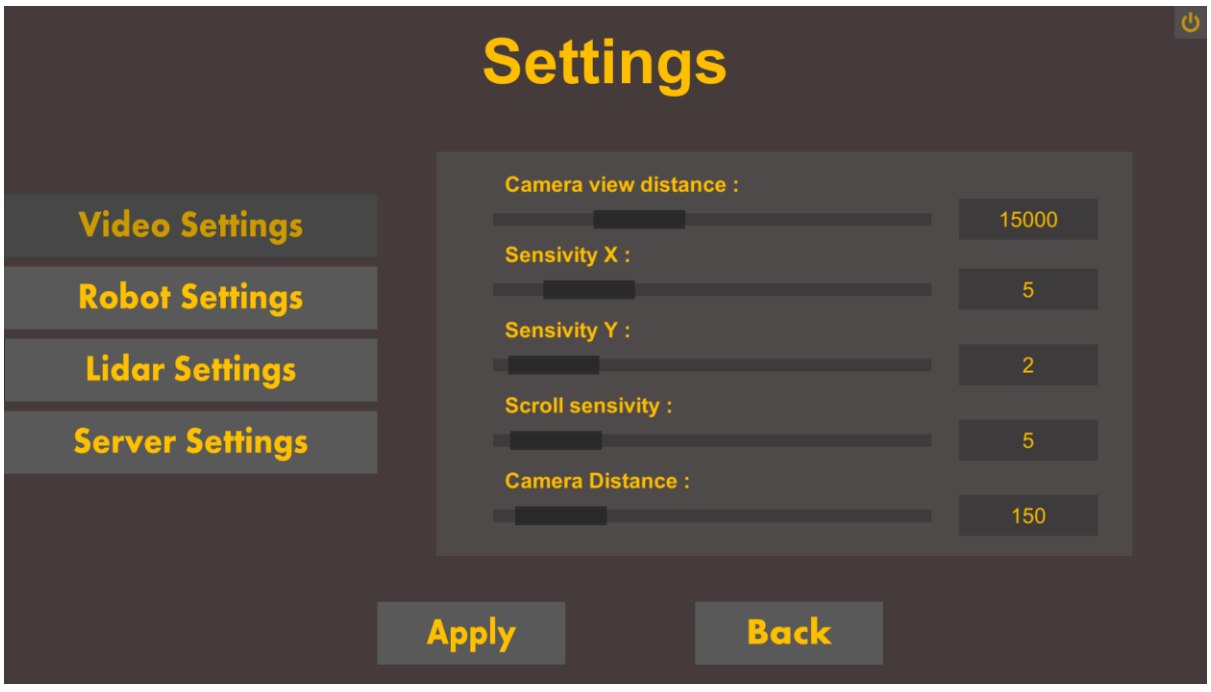


Figure 3

Les paramètres vidéo (Figure 3) permettent d'adapter la caméra du mode de simulation aux préférences de l'utilisateur. Tous les paramètres sont réglables soit grâce à la jauge qui leur est associée soit par le champ texte qui affiche aussi la valeur du paramètre.

Le premier paramètre « camera view distance » sert à changer la distance de rendu de la caméra afin de limiter la distance d'affichage pour, par exemple, faire fonctionner la simulation sur des machines avec des cartes graphiques peu puissantes.
:
Les deux paramètres suivants « sensivity X / Y » servent à régler la sensibilité des déplacements de la caméra autour du robot.

Le paramètre suivant « scroll sensivity » sert à régler la sensibilité du changement de distance de la caméra lorsque l'on utilise la molette de la souris.

Enfin, le dernier paramètre « camera distance » sert à régler la distance par défaut de la caméra.

Paramètres du robot

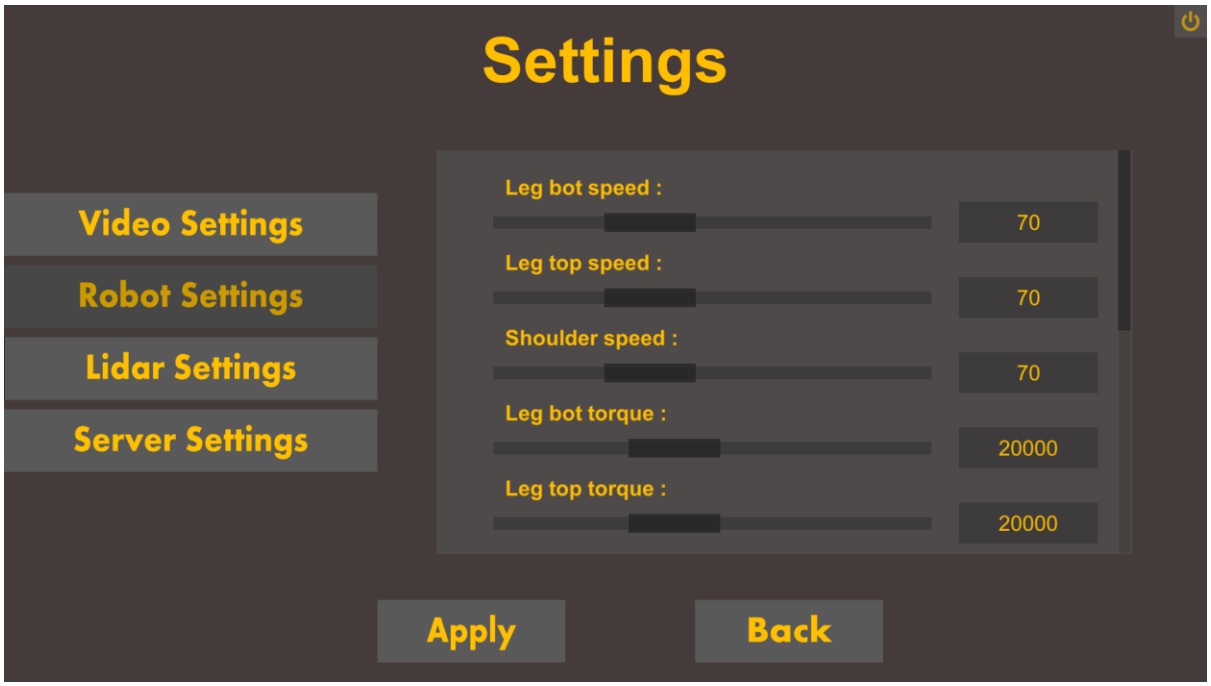


Figure 4

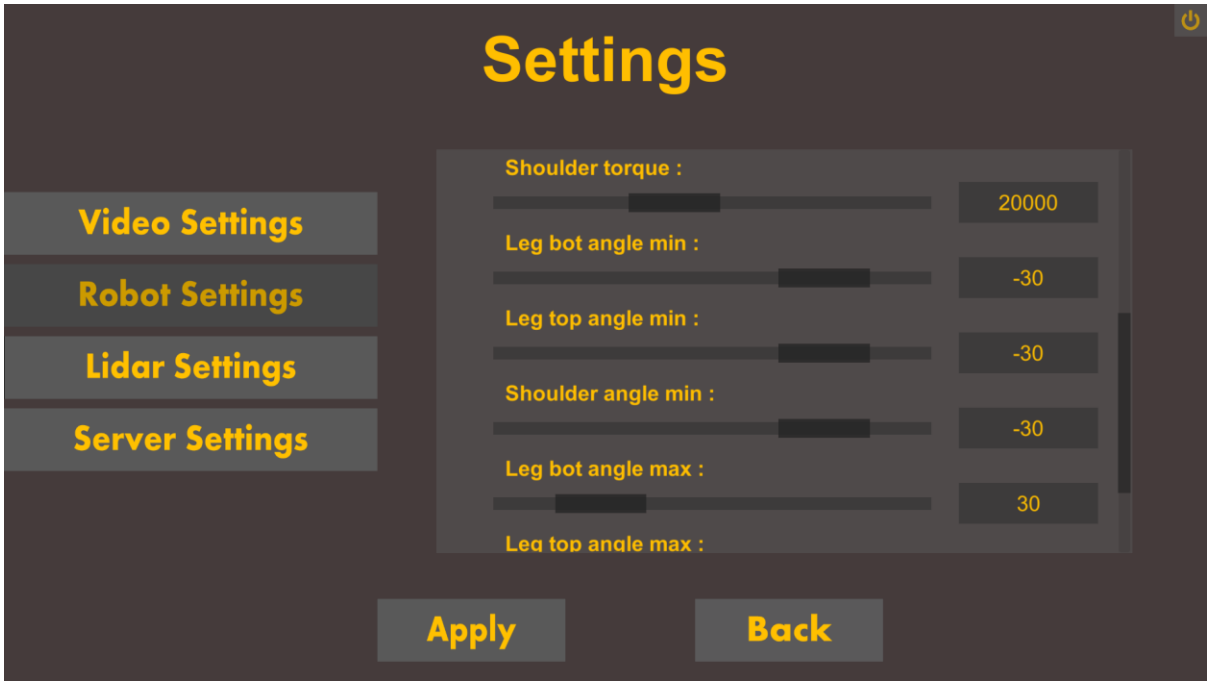


Figure 5

Figure 6

Les paramètres du robot (Figures 4, 5 et 6) permettent de régler quatre paramètres dans chacun des groupes de moteurs du robot suivant : les moteurs du genoux (Leg bot), les moteurs de la première rotation de l'épaule (Leg Top) et les moteurs de la seconde rotation de l'épaule (Shoulder).

Le premier paramètre est la vitesse de rotation du robot « speed ».

Le deuxième paramètre est le torque du moteur « torque ».

Enfin, les deux derniers paramètres « angle min et max » permettent de modifier le nombre de degrés que le moteur peut parcourir depuis sa position de repos.

Paramètres du lidar

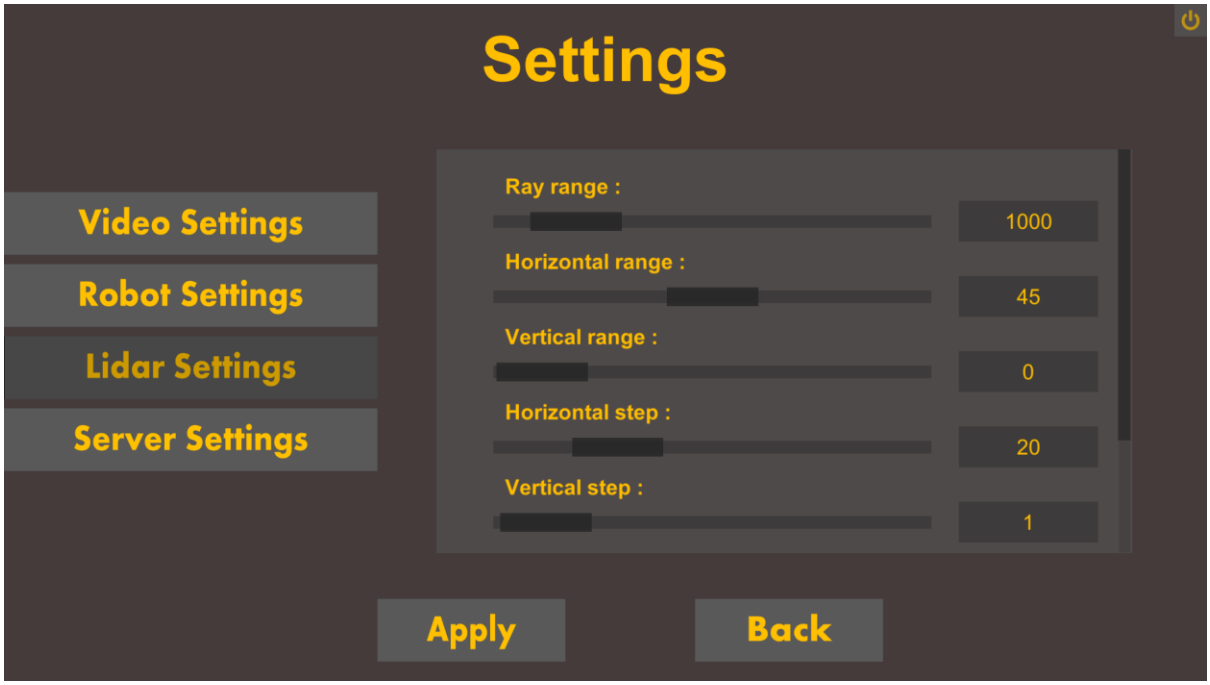


Figure 7

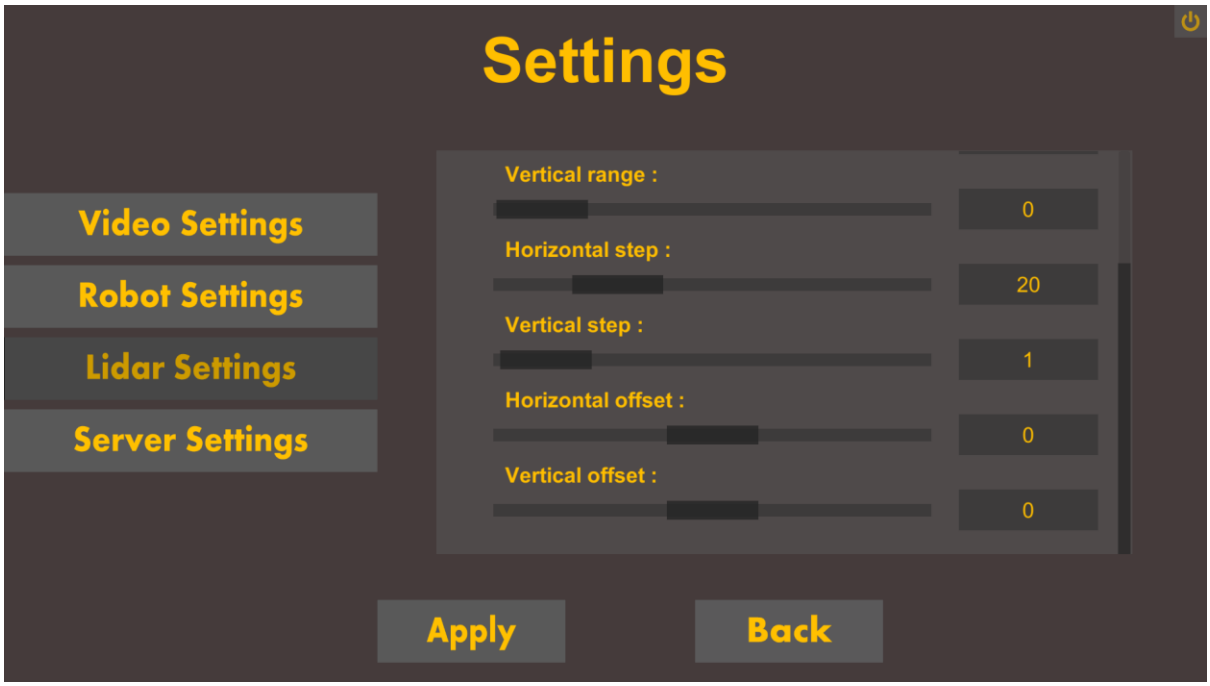


Figure 8

Les paramètres du lidar (Figures 7 et 8) permettent de modifier les fonctionnalités du lidar. Le fonctionnement de ce dernier sera expliqué dans la section dédiée au fonctionnement du simulateur.

Le premier paramètre « ray range » sert à modifier la portée d'un rayon, c'est-à-dire la distance maximale à laquelle le lidar peut détecter un obstacle.

Les deux paramètres suivants « horizontal / vertical range » permettent de régler la largeur du « champ de vision » du lidar (en degrés). Il est important de noter que la largeur du « champ de vision » sera deux fois égale à ce paramètre plus un, car le lidar « regardera » x degrés à gauche, x degrés à droite et droit devant lui. Ainsi 45 degrés représentent un champ de vision de 91 degrés (pour un maximum de 181 degrés lorsque le paramètre est à 90 bien que cela sera tronqué à 180 à cause de la nature de la formule utilisée pour simuler le lidar).

Le paramètre « horizontal / vertical step » est le pas entre chaque rayon. Ainsi un pas de 20 représente un espacement de 20 degrés entre chaque rayon. La précision maximum est de 0.1 degrés. Il est important de noter que l'on ne choisit pas directement le nombre de rayons du lidar, il faut le calculer à partir du pas et de la largeur du « champ de vision ».

Enfin, les deux derniers paramètres « horizontal / vertical offset » permettent de décaler la direction des rayons si besoin.

Paramètres du serveur

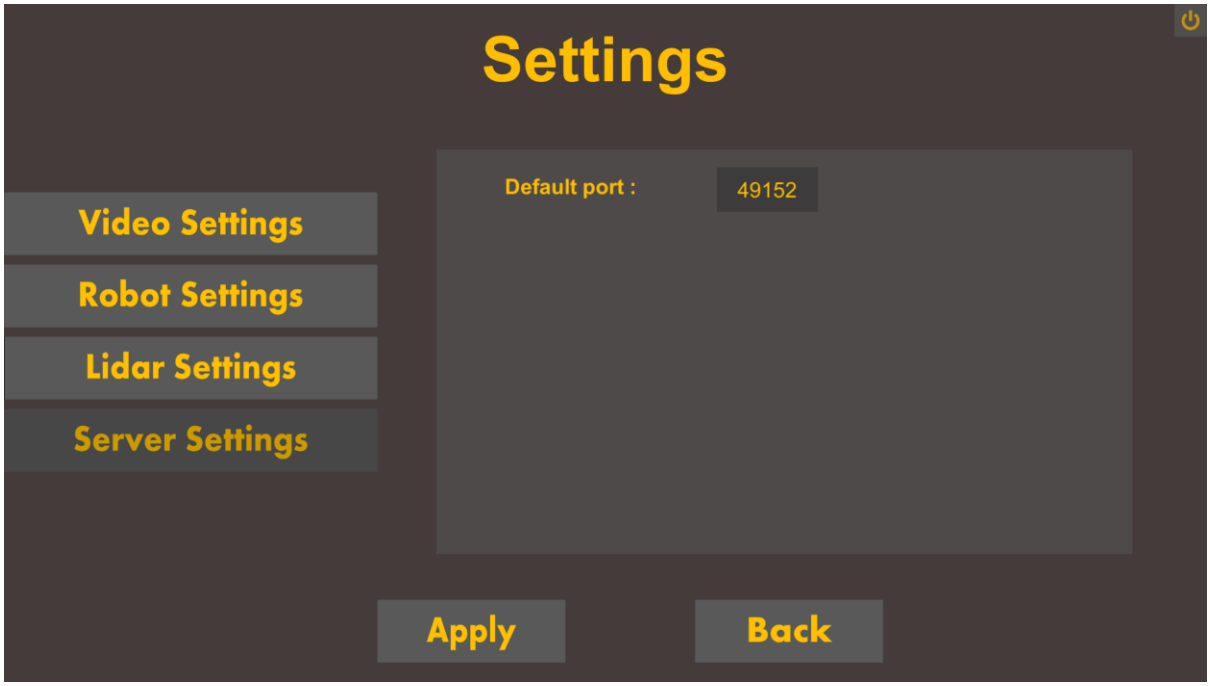


Figure 9

Les paramètres du serveur (Figure 9) ne permettent que de régler le port par défaut d'hébergement du serveur.

Ecran de sélection du terrain

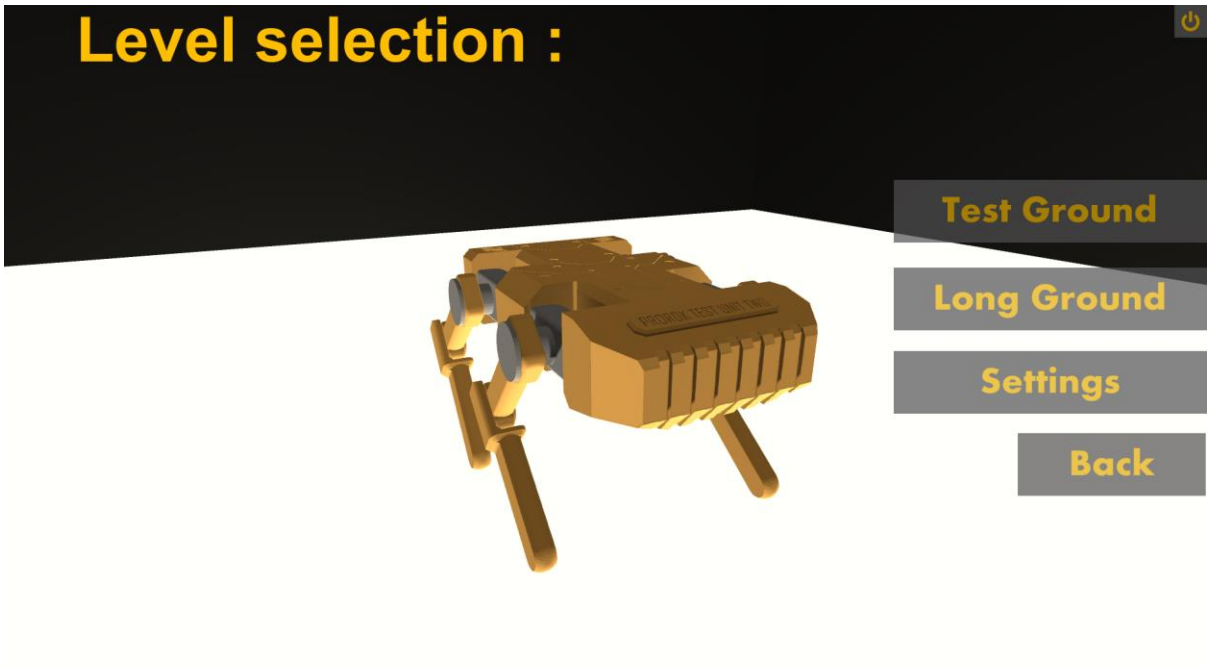


Figure 10

Quel que soit le type de simulation que l'on choisit, il faudra sélectionner le terrain sur lequel nous souhaitons entrainer le robot (Figure 10). Actuellement, deux terrains sont disponibles dans le simulateur : le terrain de test et le terrain long. Dans le cadre du projet que nous présentons ici, le choix du terrain n'a pas grande influence sur l'entrainement du robot, néanmoins nous gardons la possibilité de changer de terrain pour, à l'avenir, améliorer notre modèle.

Mode entrainement

Figure 11

Le mode d'entrainement (Figure 11) ne permet d'accéder qu'aux outils de supervision du robot et à la gestion du serveur.

En haut à gauche de l'écran se trouve le panneau de gestion du serveur, on peut y sélectionner le port d'hébergement du serveur, lancer ce dernier ou l'arrêter et en connaître le statut (Figures 12 et 13).

Figure 12

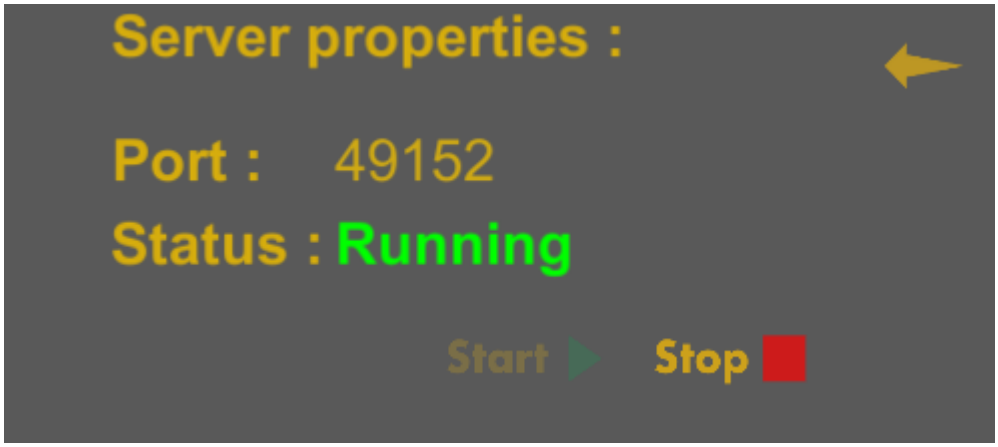


Figure 13

En bas à droite de l'écran, on peut voir le panneau de statut du robot (Figure 14).

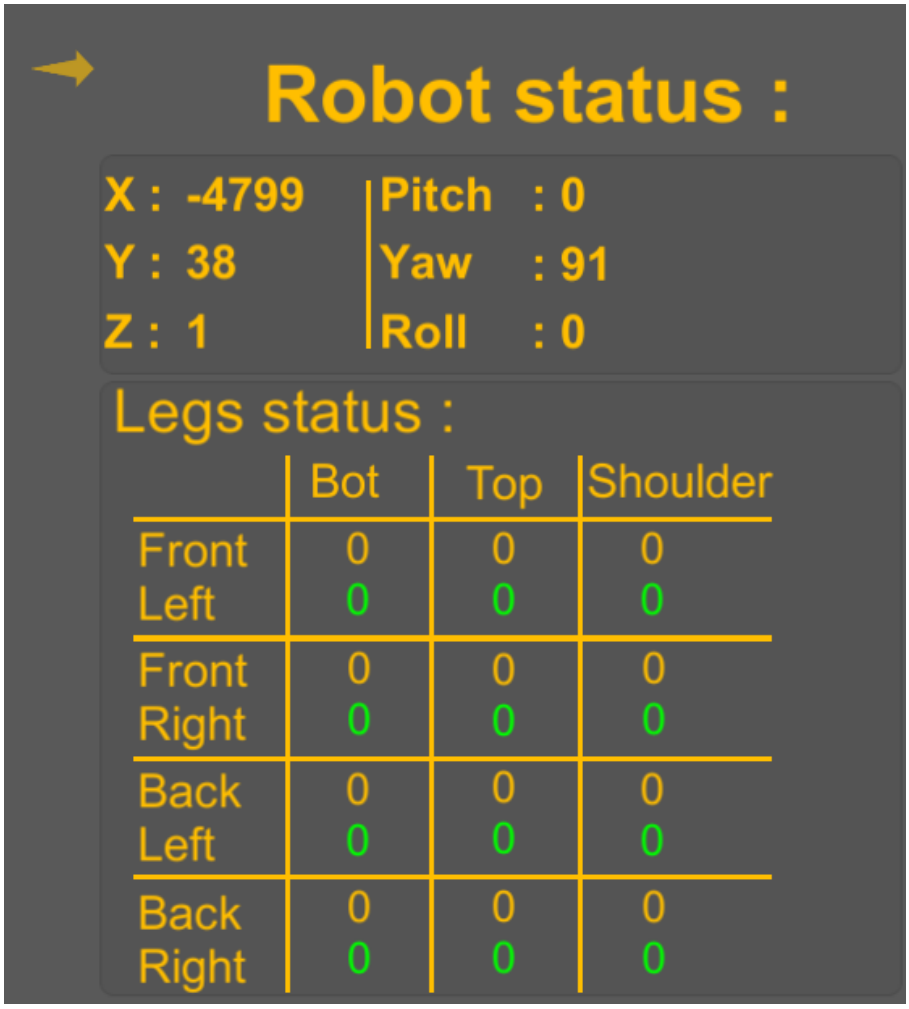


Figure 14

On y retrouve la position du robot sur le terrain, son attitude (Rotation en x, y, z) et les commandes ainsi que les positions actuelles de chaque moteur. Dans le tableau ci-dessus les chiffres en jaune représentent les commandes et les chiffres en vert

représentent les positions. Si un moteur n'est pas à la position commandée, sa position sera alors écrite en rouge afin d'avoir une meilleure visibilité de ce qui se passe.

Mode simulation

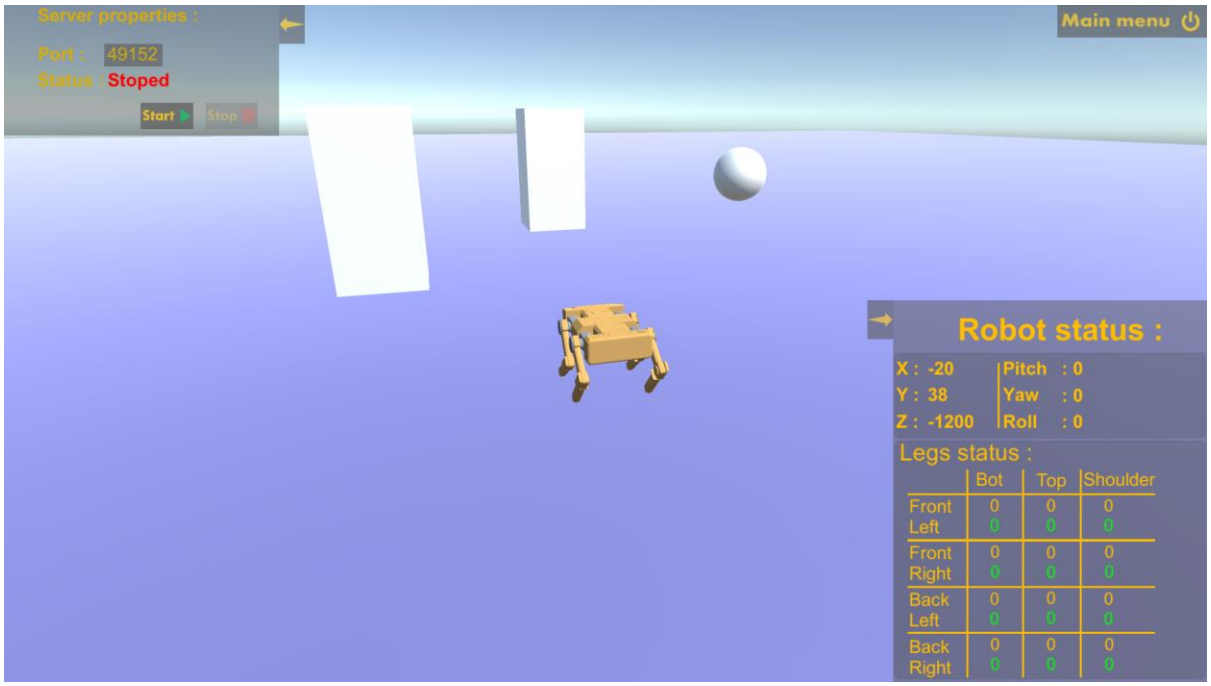


Figure 15

Ce mode propose les mêmes possibilités que le mode d'entrainement. Il permet cependant de visualiser le robot (Figure 15), de déplacer la caméra autour de ce dernier directement avec la souris (click droit + mouvement pour la rotation et molette pour la distance).

Que ce soit pour le mode d'entrainement ou le mode de simulation, il est possible de revenir au menu principal depuis le bouton situé en haut à droite de l'écran, cela coupe aussi automatiquement le serveur afin de ne pas encombrer un port inutilement, ce qui pourrait générer des bugs si on relance une simulation par la suite sans quitter l'application.

De plus, un bouton est disponible pour quitter l'application (le bouton power) où que l'on soit afin de la rendre plus ergonomique.

Sur les figures 16 et 17, deux autres vues du robot, sous un angle différent et plus proche, sont présentées. De plus les panneaux d'informations sont rétractés (grâce au bouton en forme de flèche). A noter que le panneau serveur nous montre toujours le statut de ce dernier, même retracté.

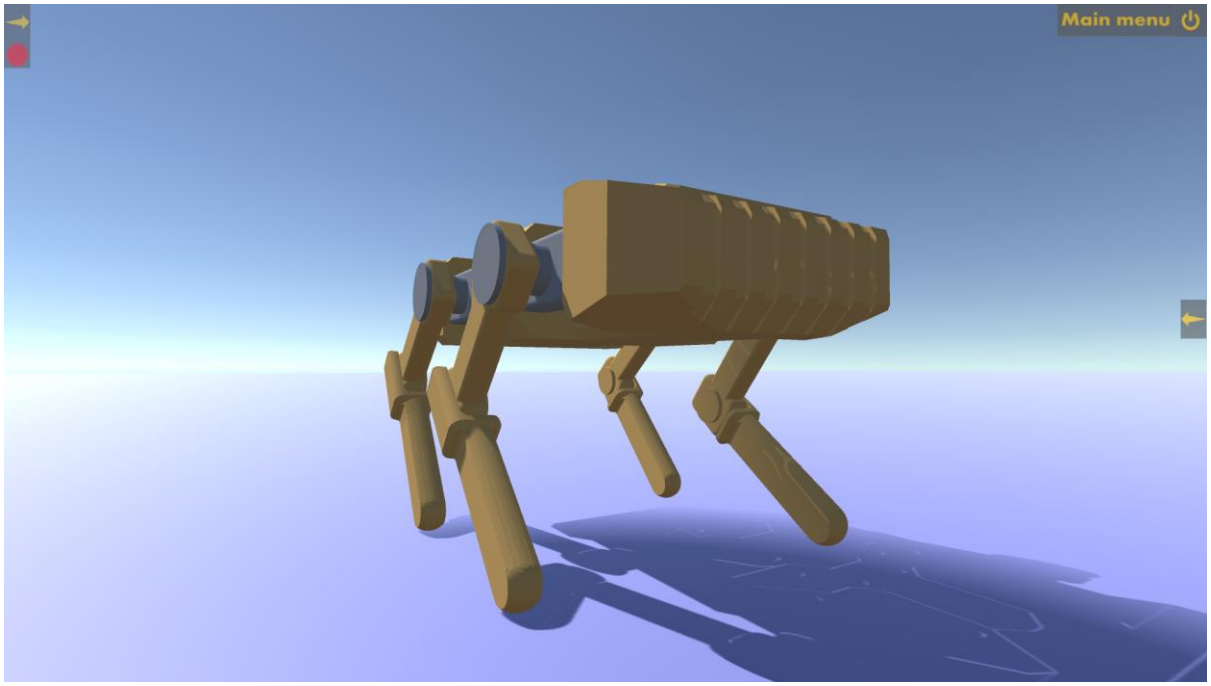


Figure 16

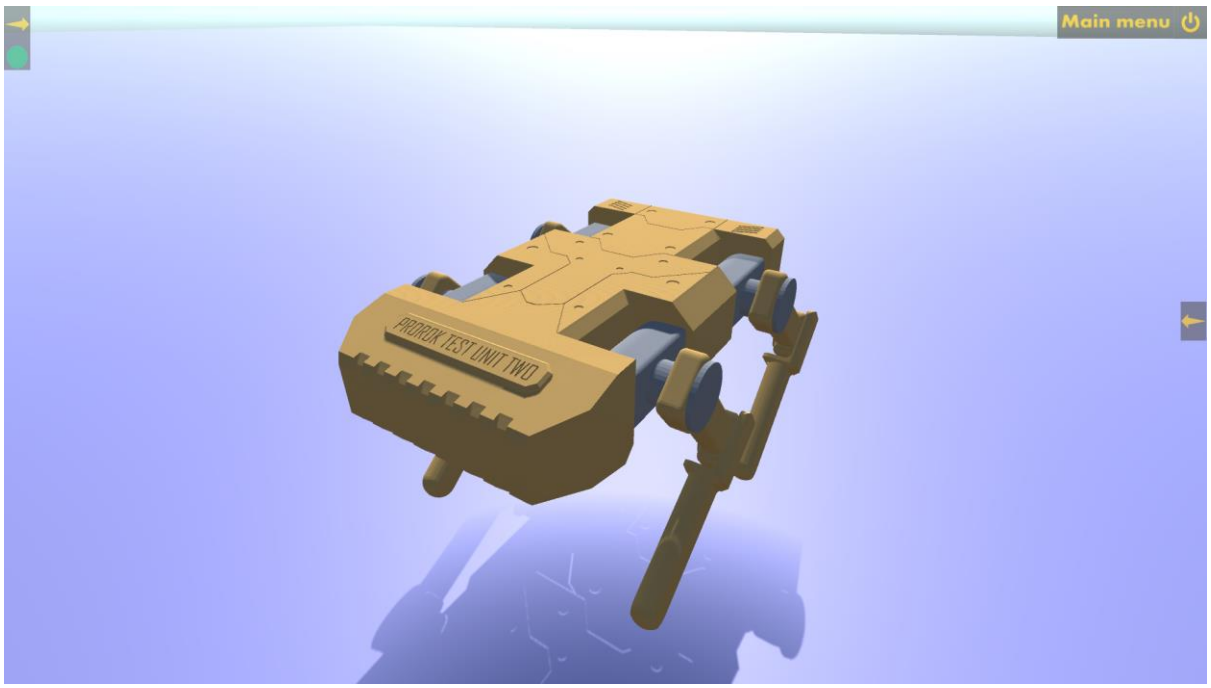


Figure 17

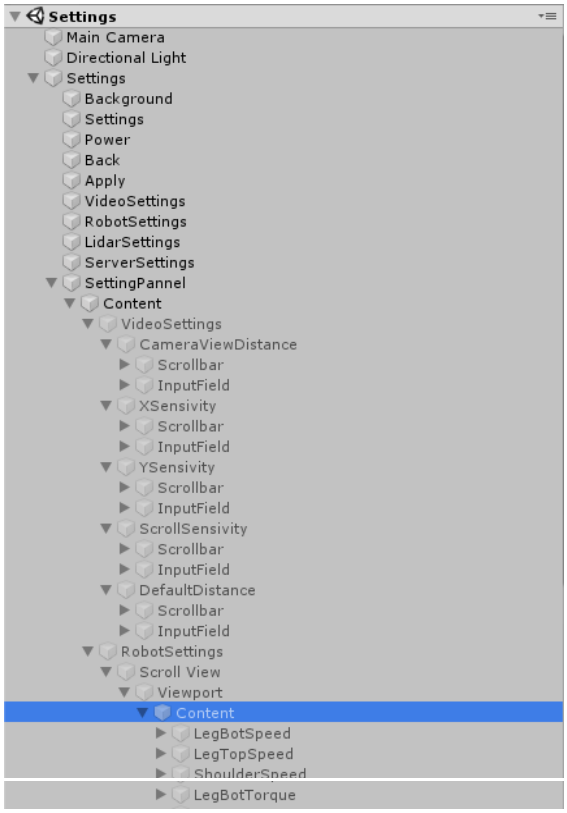
Fonctionnement du simulateur :

Dans cette partie, nous allons aborder plus en détail le fonctionnement du simulateur. La plupart des fonctionnalités du simulateur sont directement stockées dans un prefab de Unity appelé ProrokUnitTest2. Nous allons donc commencer par tout ce qui n'est pas contenu dans ce dernier à savoir les menus et les terrains.

Fonctionnement des différents menus

Les trois pages de menus, à savoir le menu principal, le menu de sélection de niveau et le menu de paramétrage fonctionnent globalement de la même façon bien que le menu de paramétrage contienne bien plus de contenu.
Chaque page est contenue dans une scène de Unity (MainMenu, Settings et LevelSelection) et sera chargée au besoin. Il aurait été possible de mettre les trois pages dans la même scène en utilisant le même système d'affichage que celui utilisé dans les paramètres mais cela aurait complexifié encore plus l'arborescence des éléments et aurait pu conduire à plus de bugs. Nous avons donc privilégié la simplicité.

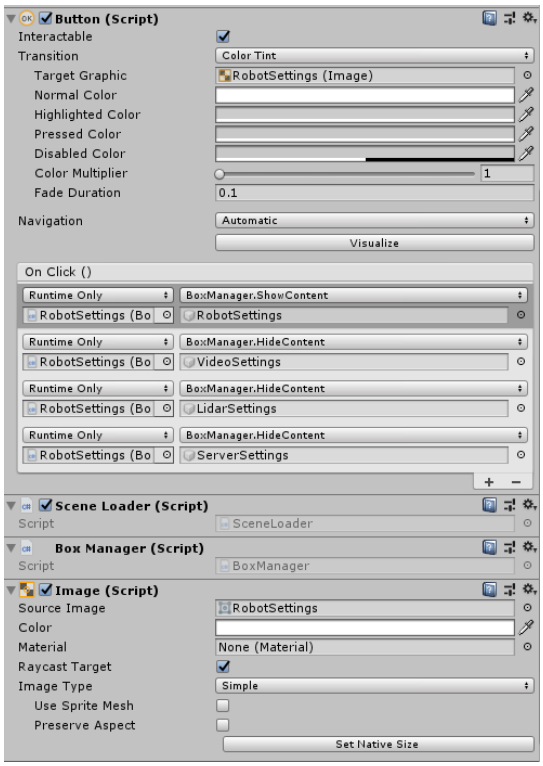
Les différents éléments affichés à l'écran sont contenus dans un composant **canvas** qui est associé à la camera principale de la scène. Ensuite, tous les composants nécessaires sont ajoutés en tant qu'enfants au **canvas**.



La figure 18 représente une partie des éléments présents dans le **canvas** du menu de paramétrage. Le **canvas** est appelé Settings (troisième composant en partant du haut). Tous les composants en dessous de settings sont des enfants de ce **canvas** et sont donc affichés à l'écran.

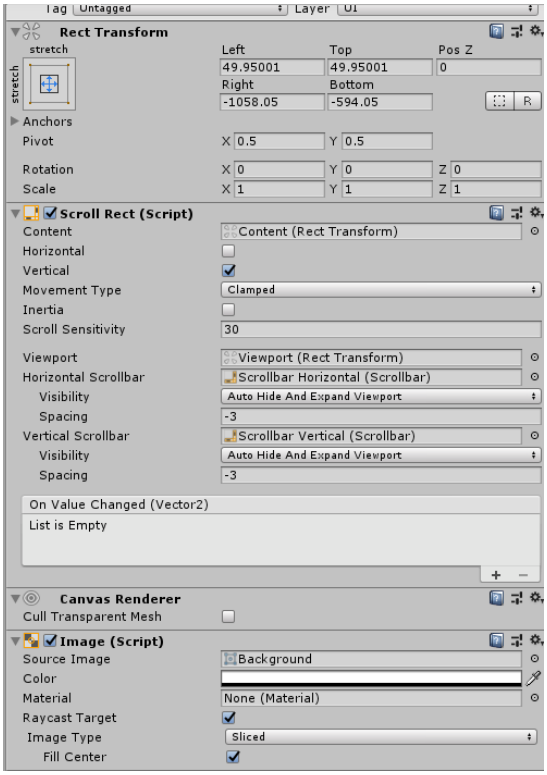
Chaque page a plusieurs types de composants qui permettent différentes interactions avec l'utilisateur.

Le premier type est le **Button**, c'est un composant simple qui permet d'exécuter une fonction lorsque l'utilisateur appuie dessus. Il est notamment utilisé pour changer de scène en appelant les différentes fonctions du fichier **SceneLoader** (contenu dans le sous dossier **UIScripts** du dossier **Scripts**). Ce script contient notamment les variables qui permettent au programme de connaitre la scène à charger mais aussi le type de simulation que l'on souhaite exécuter (mode de simulation ou mode d'entrainement). Il contient également les mutateurs et accesseurs de ces variables et enfin la fonction **SceneSwitcher** qui permet de commander à Unity de changer de scène. Il est aussi utilisé pour masquer ou afficher des composants grâce aux fonctions du script **BoxManager**. Ce dernier script permet de masquer ou d'afficher un composant en activant ou désactivant ce dernier ; composant qui est passé en paramètre. Cela permet ensuite d'automatiquement appliquer cela à tous les enfants du composant ce qui permet de simplement masquer ou afficher un panneau entier comportant de multiples enfants. C'est ce script qui est utilisé dans le panneau de paramétrage pour afficher les différentes options en fonction du bouton pressé par l'utilisateur. Enfin le bouton est parfois utilisé pour activer des fonctions spécifiques tel que le bouton power qui appelle la fonction **Quit** du script **QuitApp**, ce qui a pour effet de fermer l'application. Il y a aussi le bouton **Apply** qui appelle la fonction du même nom contenue dans le fichier **ApplySettings** et qui stocke les paramètres entrés dans le fichier de sauvegarde correspondant.



La figure 19 représente une partie des paramètres d'un composant bouton.
On y retrouve les différents paramètres qui permettent de définir l'aspect du bouton (quand il est cliqué, désactivé…) ainsi que le sprite (les sprites sont contenus dans le dossier Sprite) utilisé pour l'affichage du bouton. On y retrouve aussi la listes des différente fonctions exécutées lorsque l'évènement OnClick survient.

Ensuite, nous utilisons les composants de type **ScrollView** et **Pannel** afin de structurer l'affichage. Ces deux composants servent de conteneurs, le premier a l'avantage de permettre d'afficher des barres de défilement si le contenu dépasse de la zone d'affichage.



La figure 20 représente les paramètres d'un composant **ScrollView,** il se comporte de façon assez similaire au bouton, on peut modifier l'aspect des barres de défilement et il est possible de détecter un événement lors du changement de valeur bien que cela ne soit pas utilisé dans ce projet.

Pour finir, nous utilisons conjointement les composants **InputField** et **Scrollbar** afin de permettre à l'utilisateur de régler les paramètres de la simulation. Encore une fois l'aspect graphique de ces composants est assez similaire au bouton. Le comportement de ces composants pour l'entrée des paramètres est régi dans le script **SettingManager** qui est le plus gros script de gestion de l'interface utilisateur du projet. Il gère la plupart des comportements de la scène de paramétrage.
Chaque duo de composants est représenté par quatre variables dans le code comme on peut le voir sur le listing 1. Une variable pour chaque composant accompagnée de deux constantes qui servent de minimum et de maximum à la valeur du paramètre.
Les composants sont ensuite « Drag and drop » depuis l'éditeur dans le script qui est rattaché au **canvas** Settings (Figure 21).

```
public InputField camViewDistanceField; Set by Unity
public Scrollbar camViewDistanceScroll; Set by Unity
private const float CamViewDistanceMin = 1000f;
private const float CamViewDistanceMax = 50000f;
```

Listing 1

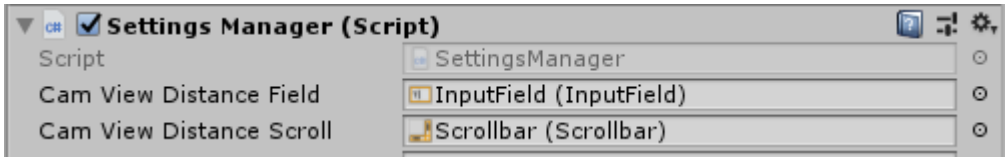


Figure 21

Les valeurs des composants graphiques sont ensuite initialisées au lancement de la scène en utilisant les paramètres du fichier de sauvegarde.

```
camViewDistanceField.text = ((int) settings.cameraViewDistance).ToString(CultureInfo.InvariantCulture);
```

Listing 2

La valeur donnée par le curseur d'une **Scrollbar** est comprise entre 0 et 1, on utilise donc les valeurs minimum et maximal pour convertir la valeur du paramètre en valeur normalisée.

```
camViewDistanceScroll.value = (settings.cameraViewDistance - CamViewDistanceMin) /
                              (CamViewDistanceMax - CamViewDistanceMin);
```

Listing 3

Ensuite à chaque frame, on va vérifier que les valeurs entrées sont correctes uniquement si l'utilisateur est entrain de manipuler un composant. Si les valeurs entrées sont incorrectes, on imposera la valeur la plus pertinente (Min, max ou 0) afin de ne pas avoir de paramètre défaillant. Enfin on rafraichira la valeur du composant non sélectionné afin que la **Scrollbar** et le **InputField** aient la même valeur (Listing 4).

En fin de frame, on va récupérer la valeur de la **Scrollbar** et la reconvertir en valeur de paramètre pour la stocker comme nouveau paramètre (Listing 5). Si on ne récupère pas directement la valeur contenue dans le champ texte c'est à cause du fait que l'on transforme la valeur en int pour l'afficher et on perd donc en précision.

Ce procédé est répété pour chaque paramètre. Une critique que l'on pourrait faire à l'égard de cette façon de faire pourrait être l'utilisation abusive du copié/collé qui augmente énormément le nombre de lignes. Cela est dû au fait que chaque composant est déclaré et placé indépendamment dans l'éditeur. Bien qu'il aurait été possible d'utiliser la sérialisation pour passer outre ce problème, nous avons préféré rester sur une méthode simple et sûre (nous en reparlerons lorsque nous aborderons le fonctionnement du robot). En effet la sérialisation nous a causé de nombreux problèmes dans d'autres parties du code.

```
if (float.TryParse(camViewDistanceField.text == "" ? "0" : camViewDistanceField.text,
        out var camViewDistanceFieldValue) &&
    camViewDistanceField.isFocused)
{
    if (camViewDistanceFieldValue > CamViewDistanceMax)
    {
        if (!camViewDistanceField.isFocused)
        {
            camViewDistanceField.text = ((int) CamViewDistanceMax).ToString(CultureInfo.InvariantCulture);
        }

        camViewDistanceScroll.value = 1;
    }
    else if (camViewDistanceFieldValue < CamDistanceMin)
    {
        if (!camViewDistanceField.isFocused)
        {
            camViewDistanceField.text = ((int) CamDistanceMin).ToString(CultureInfo.InvariantCulture);
        }

        camViewDistanceScroll.value = 0;
    }
    else
    {
        camViewDistanceScroll.value = (camViewDistanceFieldValue - CamViewDistanceMin) /
                                      (CamViewDistanceMax - CamViewDistanceMin);
    }
}
else if (camViewDistanceField.text != "")
{
    camViewDistanceField.text = ((int) settings.cameraViewDistance).ToString(CultureInfo.InvariantCulture);
}
else if (!camViewDistanceField.isFocused)
{
    camViewDistanceField.text = 0 > CamViewDistanceMin
        ? 0 < CamViewDistanceMax ? "0" : ((int) CamViewDistanceMax).ToString(CultureInfo.InvariantCulture)
        : ((int) CamViewDistanceMin).ToString(CultureInfo.InvariantCulture);
}
```

Listing 4

```
settings.cameraViewDistance = camViewDistanceScroll.value * (CamViewDistanceMax - CamViewDistanceMin) +
                              CamViewDistanceMin;
```

Listing 5

Pour finir notre visite des menus du simulateur, l'animation (esthétiquement discutable) du menu de sélection des niveaux est en fait juste le modèle 3D du robot autour duquel tourne la caméra. Cela est effectué grâce au script **CameraController** qui permet à l'utilisateur de déplacer la camera dans la simulation. Initialement cette scène ne servait qu'à tester les fonctions de ce script, nous l'avons ensuite modifiée pour en faire une animation afin de rendre le simulateur un peu plus dynamique. Le **canvas** du menu n'a pas de fond contrairement aux autres menus ce qui permet d'observer la scène en arrière-plan.

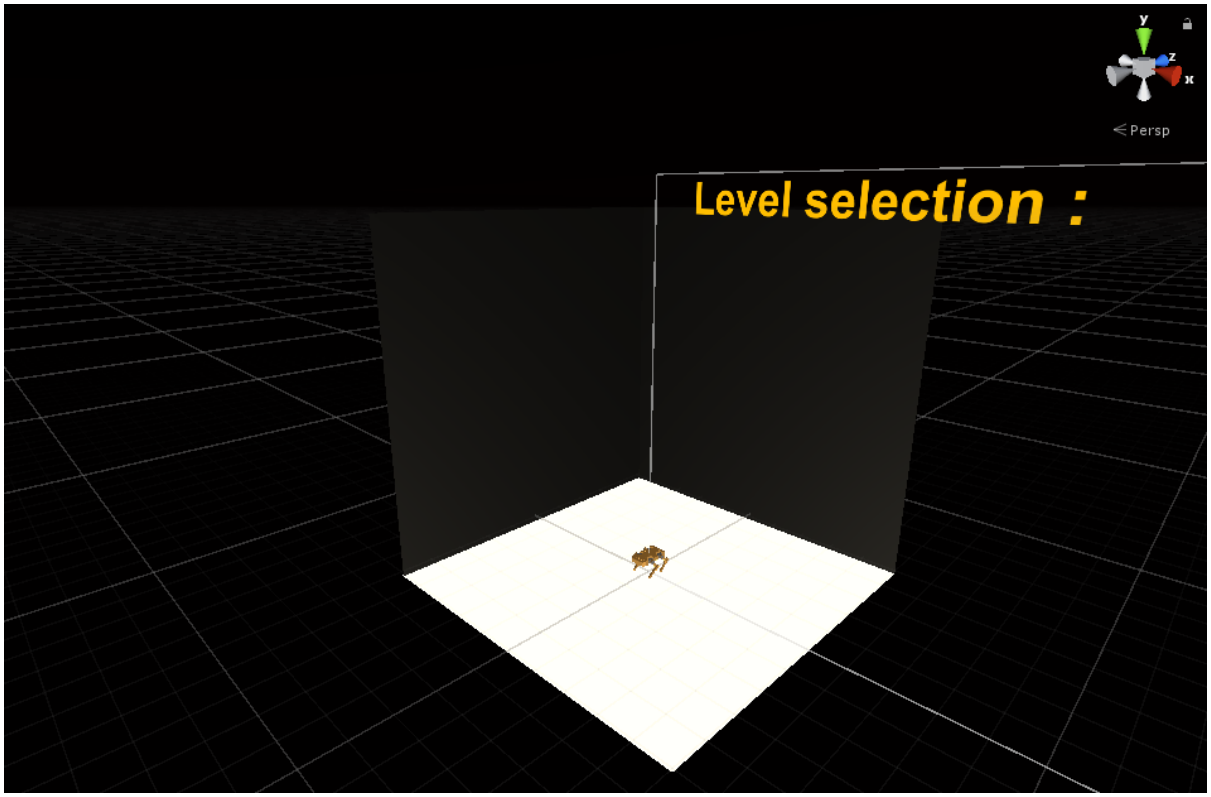


Figure 22

Fonctionnement des Terrains

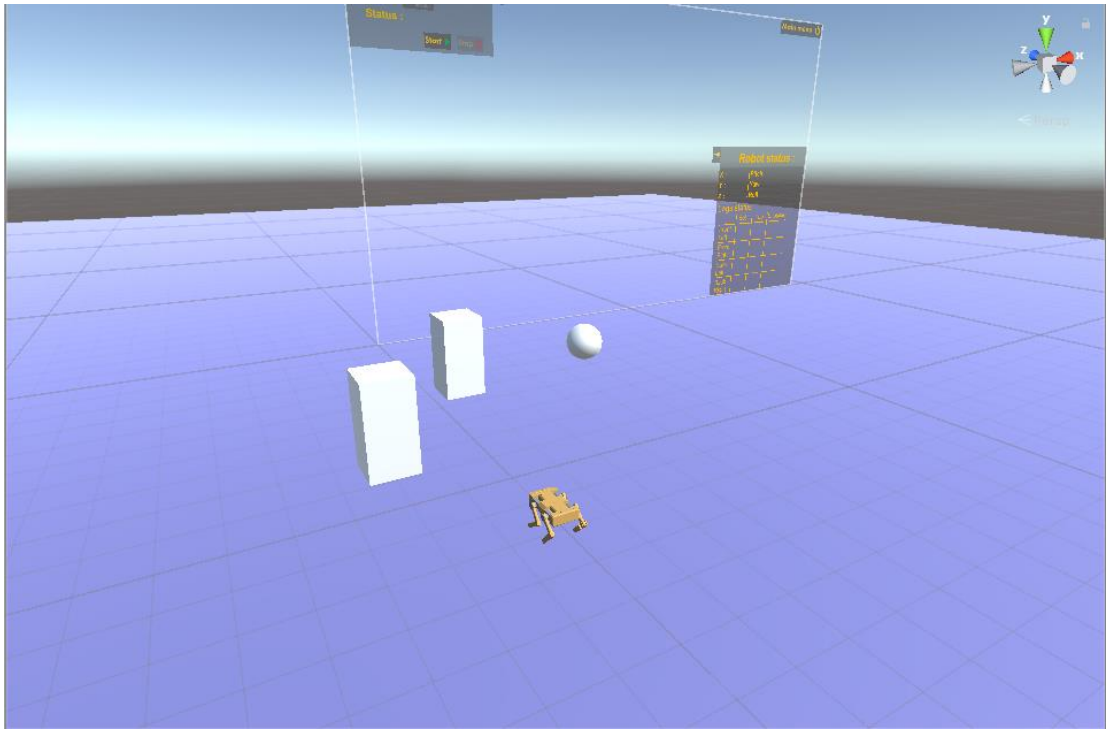


Figure 23

Les terrains sont constitués uniquement de quelques solides basiques, le plus important est le plan qui sert de sol. Ce plan appelé « Ground » est le seul solide que le robot ne considèrera pas lors du calcul du score de l'intelligence artificielle en cas de collision. En effet, tout solide qui n'a pas le nom Ground et qui déclenche une collision avec le robot, infligera une pénalité à ce dernier. Chaque solide possède un composant de type **Collider** afin de pouvoir réagir aux collisions et de pouvoir être détecté par le Lidar.
Enfin chaque terrain possède un prefab ProrokTestUnit2 qui contient tous les scripts et composants nécessaires au fonctionnement de la simulation, y compris le **canvas** qui affiche les informations sur le robot.

Fonctionnement du robot

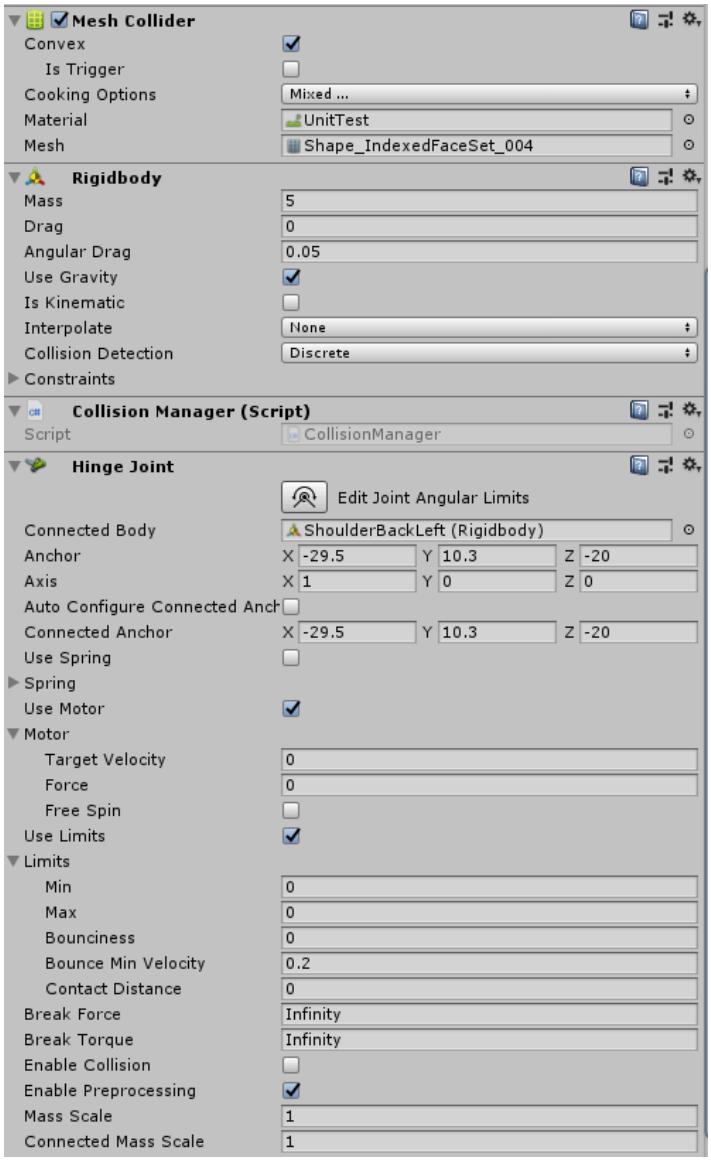
Dans cette section, nous n'aborderons que la partie « mécanique » du robot, bien que ce dernier contienne tous les autres composants, afin de simplifier la compréhension du projet. Tous les scripts de cette section et des suivantes sont directement attachés à la pièce « Body » du robot.

Le robot est en premier lieu un assemblage de solides modélisés grâce à Solidworks qui a ensuite été importé dans Blender pour pouvoir enfin l'enregistrer dans un format compréhensible par Unity. Une fois importée chaque pièce est considérée comme un enfant de l'assemblage initial ce qui permet de la garder dans un ensemble cohérent et de pouvoir manipuler l'ensemble des pièces en même temps. Ce système est très pratique pour faire des jeux vidéo mais n'était pas du tout adapté pour un simulateur où chaque pièce doit pouvoir agir sur les autres, ce qui n'est pas possible avec le système parent-enfant.

Pour résoudre ce problème, chaque pièce se voit associer des composants qui vont régir son comportement physique comme un **RigidBody** qui va gérer la masse et la gravité, par exemple, et qui est nécessaire pour appliquer d'autres composants de la physique d'Unity. Ensuite on ajoute un **MeshCollider** qui permet d'associer à la pièce une zone de collision afin qu'elle réagisse de façon réaliste (aussi réaliste que le peut Unity) lors d'un contact avec un autre solide. Enfin pour régir les interactions entre les pièces, on ajoute un **Joint** entre les pièces qui en ont besoin. Dans ce projet, nous utilisons deux types de joint le **FixedJoint** et le **HingeJoint**. Le premier est une simple contrainte entre deux pièces que l'on utilise, par exemple, pour fixer un capteur (tout au moins l'équivalent simulé) sur une pièce. Le second joint est beaucoup plus intéressant. Il fonctionne comme une charnière de porte et offre plusieurs fonctionnalités très pratiques. Il permet, par exemple, de limiter l'angle entre les deux pièces (C'est de là que viennent les paramètres angle min et angle max que l'on retrouve dans la page Settings) ou d'utiliser un système d'amortisseur qui fera rebondir les pièces si une trop grande force est appliquée sur une butée (nous ne l'utilisons pas sur ce projet). Enfin la dernière fonctionnalité de ce joint que

nous souhaitons présenter est, dans le cas de notre projet, la plus importante pour ce composant. Il s'agit de la possibilité d'appliquer une force motrice aux pièces connectées entre elles. Il faut noter que Unity fait la différence entre un déplacement et une force, même lorsque l'objet en mouvement possède un **RigidBody** avec une masse. Ainsi, une simple rotation de la jambe ne suffirait pas à déplacer le robot, c'est pour cela que le composant moteur du **HingeJoint** est très important. Il nous permet donc de pouvoir déplacer notre robot sans avoir à recoder nous-mêmes une fonction pour appliquer une force (ce qui devient très vite compliqué sur Unity car les rotations utilisent des nombres complexes ce qui les rend difficile à manipuler). Vous retrouverez, en figure 24, une image représentant les différentes pièces du robot et un exemple des composants attachés à une pièce de ce robot (Figure 25).

La figure 24 liste le,s pièces du robot. A noter que toutes les pièces sont les enfants d'un objet appelé pivot. Pour résumer rapidement, car cela n'a pas d'incidence sur la compréhension du projet, lors de l'importation de l'assemblage 3D, toutes les pièces prennent pour origine l'origine de l'assemblage. Ainsi si on appliquait une rotation à une pièce cette dernière tournerait de façon totalement incohérente. En ajoutant un point de pivot, l'origine des pièces enfants deviennent ce point de pivot et les pièces tournent à nouveau de façon cohérente. Cela n'aurait pas été problématique en l'état actuel du projet mais cela a beaucoup aidé lors du début du projet et des phases de débogage, c'est donc pour cela que ces points de pivot sont toujours présents.



On retrouve sur la figure 25 les principaux composants dont nous avons parlé plus haut. Le script de Collision détecte les collisions et sert uniquement pour le calcul du score. A noter que nous n'avons pas parlé volontairement de plusieurs autres composants de chaque pièce qui ne servent qu'au rendu visuel et qui ne sont, par conséquent, pas très intéressants pour la compréhension du projet. Toutefois, il est important de savoir que la couleur, la gestion de la lumière et même la résistance des matériaux ont dû être spécifiées afin d'avoir le robot que l'on voit à l'écran. Sans cela le robot ne pourrait, par exemple, pas avancer car le matériau de base se comporte comme de la glace ce qui fait déraper le robot sur le sol.

Maintenant que nous avons parlé de ce qui permet de rendre le robot physiquement crédible, nous allons nous intéresser au code qui lui permet de fonctionner.

La majorité du comportement du robot est codé dans le script **Controller** (dossier Script) qui fait appel à d'autres petits scripts pour fonctionner. Ce script communique avec le reste des fonctionnalités *via* le script **Manager** qui fait office de messager entre tous les différents scripts. Le script Controller, étant le script principal du projet, a connu énormément de changements au cours du développement. Il a donc été nécessaire d'élaborer un script intermédiaire pour normaliser les informations transmises afin de ne pas perdre de fonctionnalités au fur et à mesure des changements.

Les composants physiques du robot sont stockés dans la variable prorokTestUnit2. Cette variable est composée de quatre objets **Leg** qui sont chacun composé de trois objets **Motor.** Un objet **Motor** permet de stocker le composant **HingeJoint** dont nous avons parlé plus haut ainsi que plusieurs paramètres pour son paramétrage.

On peut voir sur le listing 6 les attributs de l'objet **Motor**.

```
[System.Serializable]
Scripting component | 5 usages | CDulouard | 3 exposing APIs
public struct Motor
{
    public string name; Set by Unity
    [Range(-180, 180)] public float targetPosition; Set by Unity
    public GameObject motor; /*    The GameObject we want to move    */ Set by Unity
    public GameObject sensor; Set by Unity
    [Range(-180, 180)] public float angleMin; Set by Unity
    [Range(-180, 180)] public float angleMax; Set by Unity
    public float speed; Set by Unity
    public float torque; Set by Unity

    private HingeJoint _motorJoint; /*    The HingeJoint of the GameObject    */
```

Listing 6

Les méthodes de cet objet sont uniquement des mutateurs et accesseurs, c'est pour cela que nous n'en parlerons pas. La première instruction est destinée à l'éditeur d'Unity. Cela sert à sérialiser les objets comme nous en avions parlé précédemment lors de la présentation des paramètres. Cela permet de pouvoir directement récupérer tous les objets de ce type dans une liste depuis l'éditeur, ce qui aurait amélioré l'efficacité du code. Malheureusement cela a entrainé, à de nombreuses reprises, la perte de tous les paramètres du robot (ce qui entraine à chaque fois jusqu'à une heure de perte de temps à tout réparer). C'est pour cela que finalement les objets **Motor** ont été encapsulés dans des objets **Leg** puis **ProrokTestUnit2** afin que cela ne se reproduise pas. C'est aussi pour cela que les paramètres n'ont pas hérité de cette méthode afin d'alléger le code.
Dans les Attributs des moteurs, on peut remarquer que trois variables ont été grisées car non utilisées. Cela est dû au fait que nous les manipulons directement sans passer par des mutateurs et accesseurs.
Enfin une remarque pourrait être faite sur le fait que la plupart des attributs sont publiques. Cela est dû au fait que nous avons opté pour la même logique orientée composants que Unity. Etant donné que beaucoup de composants sont liés sans pour autant passer par de l'héritage, cela permet de récupérer facilement des valeurs entre elles sans devoir encombrer le code de nombreux mutateurs et accesseurs.

Nous n'aborderons pas les objets **Leg** et **ProrokUnitTest2**, ce sont de simples dictionnaires qui possèdent juste la possibilité de récupérer facilement les valeurs des moteurs.

A présent que nous avons vu comment sont stockés les objets physiques dont on a besoin, revenons au script **Controller**. Pour le contrôle du robot, trois fonctions sont utilisées :
- **InitRobot**
- **RefreshMotors**
- **RefreshSensorValues**

InitRobot a pour but d'appliquer les bons paramètres aux **HingeJoint** du robot (comme la vitesse et le torque).
Elle appelle en premier la fonction **ImportMotorsSettings** qui lit le fichier de paramètres pour récupérer et stocker les paramétrages pour chaque objet.
Ensuite elle appelle la fonction **InitMotors** qui appelle pour chaque moteur la fonction **InitMotor** qui se charge d'appliquer les paramètres à chaque moteur.

```
private static HingeJoint InitMotor(GameObject motor, float xMin, float xMax)
{
    /*  return the HingeJoint of the GameObject with its limits  */
    var joint = motor.GetComponent<HingeJoint>();
    var limits = joint.limits;
    limits.max = xMax;
    limits.min = xMin;
    joint.limits = limits;
    return joint;
}
```

Listing 7

Comme on le voit sur le listing 7, la fonction se contente de retourner un joint correctement paramétré que l'on appliquera ensuite au moteur concerné.
La fonction d'application des paramètres fonctionne de manière similaire.

La fonction **RefreshMotors** appelle pour chaque moteur la fonction **SetMotorPosition** que l'on peut voir dans le listing 9. Pour comprendre le fonctionnement de la fonction il faut d'abord expliquer le fonctionnement du composant moteur du **HingeJoint**. Ce composant fonctionne plus ou moins comme un moteur à courant continu. On lui donne une vitesse positive ou négative et le composant va tourner dans un sens ou un autre à la vitesse indiquée. Or pour ce projet nous souhaitons piloter le moteur comme un servo-moteur, c'est-à-dire en lui donnant un angle à tenir. Pour cela la fonction va regarder le sens dans lequel doit tourner le moteur pour atteindre sa position, va lui assigner une vitesse égale à plus ou moins la vitesse que nous avons spécifiée dans les paramètres. Si le moteur atteint une fourchette autour de la position (plus ou moins 0.15 degrés) alors on lui assigne une vitesse nulle tout en maintenant le même torque afin que le moteur ne

bouge pas de la position. On met aussi des limites de rotation très proches de la position pour être sûr que le robot ne bouge pas de sa position. Enfin la fonction retourne le moteur avec les bons paramètres afin de l'assigner au **HingeJoint** concerné.

```
private static void SetMotorPosition(HingeJoint motor, float position, int speed, int torque, float xMin,
    float xMax)
{
    /*    Use the motor component of the HingeJoint to put the GameObject to the right angle.
     *    It take the HingeJoint of the GameObject, the angle of the sensor attached to the pivot point,
     *    the target position, the speed of the rotation and the torque of the motor as input.
     *    The target position is modulo 180 so the input can be over 180 or less than -180
     */
    var limits = motor.limits;
    position %= 180;
    var newMotor = motor.motor;
    newMotor.force = torque;
    var angle = motor.angle;
    if (((int) position).Equals((int) angle))
    {
        /*    Stop the motor at the target position    */
        if ((int) limits.max != (int) position & (int) limits.min != (int) position)
        {
            /*    Set new limit to avoid any movements    */
            limits.max = (int) position + 0.15f;
            limits.min = (int) position - 0.15f;
            motor.limits = limits;
        }

        newMotor.targetVelocity = 0;
    }
    else
    {
        /*    Move the motor to the target position    */
        if ((int) limits.max != (int) xMax & (int) limits.min != (int) xMin)
        {
            /*    Set new limits if the motor was blocked    */
            limits.max = xMax;
            limits.min = xMin;
            motor.limits = limits;
        }

        /*    Chose the right velocity to reach the target position    */
        newMotor.targetVelocity = position < angle ? -speed : speed;
    }

    motor.motor = newMotor;
}
```

Listing 8

La fonction **RefreshSensorValues** sert juste à remplir un dictionnaire avec tous les paramètres importants du robot à transmettre aux autres fonctions et à l'intelligence artificielle qui pilotera le robot.

```
private void RefreshSensorValues()
{
    /*    Stores the new attitude values in _sensorValues    */
    var newValues = new Dictionary<string, float>();
    var sensorTransform = generalPurposeSensor.transform;
    var sensorTransformPosition = sensorTransform.position;
    sensorTransform.rotation.ToAngleAxis(out var angle, out var axis);
    newValues.Add("roll", angle * axis.z);
    newValues.Add("pitch", angle * axis.x);
    newValues.Add("yaw", angle * axis.y);
    newValues.Add("posX", sensorTransformPosition.x);
    newValues.Add("posY", sensorTransformPosition.y);
    newValues.Add("posZ", sensorTransformPosition.z);
    _sensorValues = newValues;
}
```

Listing 9

Nous avons à présent vu les fonctions qui gèrent le robot. La fonction **InitRobot** appelée uniquement lors de la première frame et les deux suivantes sont appelées à chaque frame afin de simuler le robot au plus proche de la réalité. Si vous regardez le script Controller, vous trouverez cependant d'autres fonctions. Certaines servent au calcul du score que nous verrons plus tard lorsque nous parlerons de l'IA et d'autres servent de tests ou pour le débogage et ne sont donc pas importantes à aborder dans ce rapport.

Fonctionnement du lidar

Nous avons déjà parlé paramétrage du lidar lors de la présentation du simulateur, cette section va donc aborder plus en détail son fonctionnement.
Le code du lidar est contenu dans le script **Lidar** et utilise la classe **LidarPoint** pour enregistrer chaque point de mesure.

```
public LidarPoint(float hAngle, float vAngle, float distance)
{
    HorizontalAngle = hAngle;
    VerticalAngle = vAngle;
    Distance = distance;
}
```

Le listing 10 représente le constructeur de la classe **LidarPoint**. Cette classe permet de stocker les coordonnées du point et la distance à laquelle se trouve un obstacle.

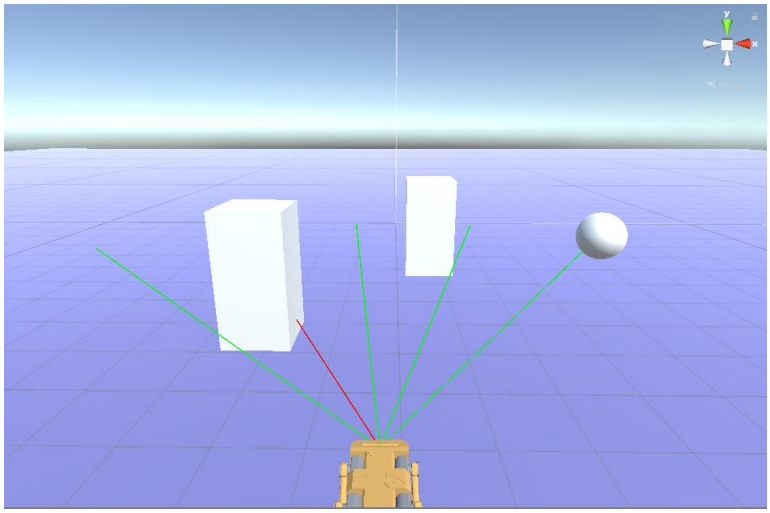
Les autres méthodes sont des mutateurs.et accesseurs.

La méthode principale du lidar est la méthode **Scan**. Elle est appelée à chaque frame afin de rafraichir les données du lidar.

```
private IEnumerable<LidarPoint> Scan()
{
    /*    Return all the point measured by the Lidar. Each point contain the horizontal angle,
     *    the vertical angle and the distance between the Lidar and the hit object.
     */
    var measures = new List<LidarPoint>();
    var position = lidar.transform.position;
    var lidarDirection = Quaternion.Euler( x: verticalOffset,  y: horizontalOffset,  z: 0) * lidar.transform.forward;

    for (float hAngle = -horizontalRange; hAngle <= horizontalRange; hAngle += horizontalStep)
    {
        for (float vAngle = -verticalRange; vAngle <= verticalRange; vAngle += verticalStep)
        {
            var direction = Quaternion.Euler( x: vAngle,  y: hAngle,  z: 0) * lidarDirection;
            if (Physics.Raycast( origin: position, direction, out var hit, rayRange))
            {
                Debug.DrawRay( start: position,  dir: direction * hit.distance, Color.red);
            }
            else
            {
                Debug.DrawRay( start: position,  dir: direction * rayRange, Color.green);
            }
            measures.Add(new LidarPoint(hAngle, vAngle, hit.distance));
        }
    }
    return measures;
}
```

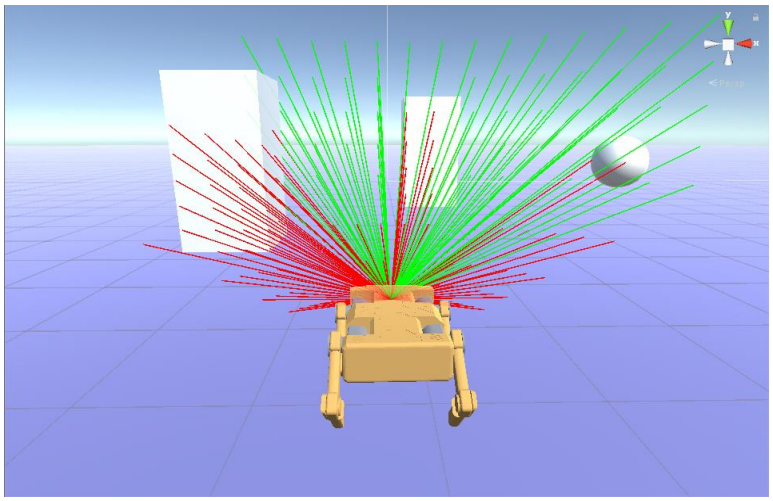
Listing 11

Comme on peut le voir dans le listing 11, la fonction **Scan** est composée de deux boucles imbriquées, une pour les mesures horizontales et une pour les mesures verticales. On commence par créer une variable qui contient la direction en face du lidar plus d'éventuels offsets (un quaternion sert à définir une rotation, utiliser la fonction **Euler** permet de travailler avec des angles d'Euler plutôt que de travailler avec des nombres complexes). Ensuite on va effectuer des mesures entre les bornes qui ont été fixées en paramètre avec un pas ayant la valeur fixée en paramètre. Pour faire ces mesures, on crée un **Raycast** à partir du lidar jusqu'à la distance fixée en paramètre et dans la direction préalablement calculée à laquelle on ajoute un décalage. Ce **Raycast** va stocker dans la variable hit l'objet qu'il touche s'il en touche un. De cela on peut déduire les coordonnées de chaque point ainsi que la distance des objets. Enfin on retourne le tableau contenant toutes les mesures. Vous trouverez en figures 26, 27 et 28 des exemples de configuration de lidar vus depuis l'éditeur de Unity.



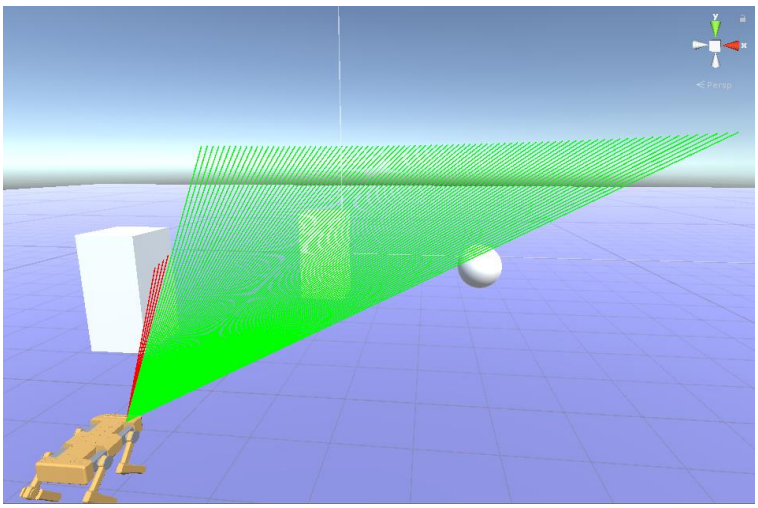
Réglages :

RayRange = 1000
Hrange = 45
Vrange = 0
Hstep = 20
Vstep = 1
Hoffset = 0
Voffset = 0



Réglages :

RayRange = 1000
Hrange = 45
Vrange = 20
Hstep = 5
Vstep = 5
Hoffset = 0
Voffset = 0



Réglages :

RayRange = 1000
Hrange = 45
Vrange = 0
Hstep = 1
Vstep = 1
Hoffset = 20
Voffset = -20

Les figures 26, 27 et 28 sont obtenues depuis l'éditeur d'Unity et ne sont pas possibles à obtenir dans le simulateur compilé. Comme vous pouvez l'observer, les rayons détectent bien les objets (Les rayons qui touchent un objet sont rouges, le sol est considéré comme un objet). Les données récupérées sont stockées dans un

tableau dynamique, en voici un exemple : [[-45, 0 ,242,4433], [-25, 0 ,302,7066], [-5, 0 ,302,7065], [15, 0 ,302,7066], [35, 0 ,302,7066]]

Fonctionnement de l'UI

Le fonctionnement de l'UI du robot fonctionne de la même façon que les menus que nous avons vus précédemment. Un **canvas** est associé au prefab **ProrokTestUnit2**. Sur ce **canvas** on ajoute différents composants comme l'interface serveur et le panneau de statut du robot. Vous pouvez voir sur la figure 29 un aperçu de l'arborescence de ce **canvas** ainsi qu'un aperçu de ce dernier depuis la vue 2D de l'éditeur sur la figure 30.

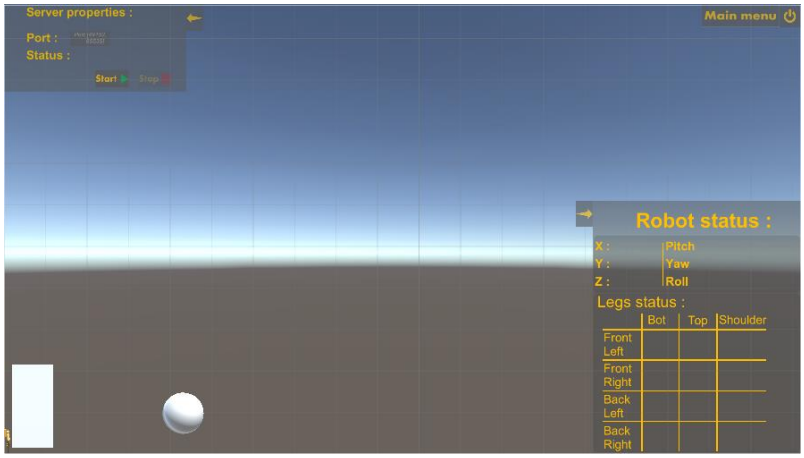


Figure 30

En plus des deux boutons en haut à droite du **canvas**, il y a deux conteneurs principaux, l'interface de serveur et l'interface de statut du robot. Chacun des deux conteneurs peuvent être réduits en appuyant sur la flèche à l'aide des mêmes fonctions que l'on a vues lors de la présentation des paramètres.

Le conteneur de propriétés du serveur contient deux boutons qui permettent de démarrer ou d'éteindre le serveur, un champ de texte pour entrer le port du serveur et divers champs textes dont un qui sert à afficher l'état du serveur (sur la figure 30 l'état du serveur n'est pas affiché car nous sommes dans l'éditeur et le jeu n'est pas lancé. Nous vous invitons à vous référer à la présentation du simulateur si vous souhaitez un aperçu des différents indicateurs). Lorsque le serveur est éteint, le bouton start est cliquable et le bouton stop est désactivé, il se passe l'inverse

lorsque le serveur est allumé. Nous rentrerons plus dans les détails dans la partie suivante traitant du fonctionnement du serveur.
Le conteneur de statut du robot est composé uniquement de champs de texte. Ceux que l'on voit ci-dessus sont les champs constants qui servent à indiquer à l'utilisateur quelle variable est affichée.

Sur la figure 31, vous pouvez voir les variables qui sont affichées lorsque le simulateur est lancé.

Le comportement de ce conteneur est codé dans le script **LegsStatusManager** et le script **AttitudeDisplayManager** qui récupèrent tous les champs textes depuis l'éditeur et y affichent les valeurs voulues.

Cette première ligne de code (listing 12) permet d'afficher la consigne pour un champ texte. La consigne est stockée dans la variable **TargetPositions** du script manager.

```
frontLeftBotTarget.text =
    Manager.TargetPositions.legFrontLeftBot.ToString(CultureInfo.InvariantCulture);
```

Listing 12

Sur le listing 13, on peut voir l'affichage de la position actuelle. Dans un premier temps, on affiche la valeur que l'on récupère dans la variable **robotDatas**, elle-même récupérée dans le script **Controller** grâce à la fonction **GetProrokUnitTest2**. On change ensuite la couleur de l'affichage en fonction de la commande (vert = atteinte ou rouge = non atteinte).

```
frontLeftBotPosition.text =
    ((int) robotDatas.frontLeft.legBot.GetAngle()).ToString(CultureInfo.InvariantCulture);
frontLeftBotPosition.color =
    (int) robotDatas.frontLeft.legBot.GetAngle() == (int) Manager.TargetPositions.legFrontLeftBot
        ? Color.green
        : Color.red;
```

Listing 13

Ce code est répété pour chaque champ à afficher, seul le nom de la variable change.

Le script **AttitudeDisplayManager** fonctionne sur le même principe, nous n'en parlerons donc pas ici.

Fonctionnement du Serveur

Le fonctionnement du serveur est régi par deux scripts, le script **Server** et le script **UdpSocket**. Le script **Server** nous met à disposition deux fonctions **SetupServer** et **StopServer** qui sont appelées lorsque l'on appuie sur les boutons start et stop de l'interface de commande du serveur.

Figure 32

La fonction **Start** crée un nouveau **UdpSocket** qui écoutera sur le port spécifié dans l'interface de commande. Ensuite si le serveur est démarré, la fonction met à jour le flag **isActive** qui permet à tous les scripts de connaitre l'état du serveur (par exemple pour savoir quel bouton activer sur l'interface).
La fonction **Stop** éteint le serveur et met à jour le flag **isActive**.

Le reste du comportement du serveur est géré dans le script **UdpSocket**. Cette classe propose plusieurs méthodes, dont les méthodes de démarrage et d'arrêt du serveur qui sont utilisées dans le script **Server**. Comme on le voit sur les listings 14 et 15, le démarrage du serveur se résume au paramétrage d'un objet Socket préalablement défini, puis à la liaison du socket au port que l'on a défini et au lancement de la fonction **Receive**. Cette fonction va gérer les différents messages envoyés par le client distant.

```
private Socket _socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
```

Listing 14

```
public void StartServer(string address, int port)
{

    _socket.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.ReuseAddress,  optionValue: true);
    _socket.Bind(new IPEndPoint(IPAddress.Parse(address), port));
    Receive();
}
```

Listing 15

Sur le listing 16 se trouve le début de la fonction **Receive**. Ce code permet de récupérer les messages envoyés, de les stocker et de les traiter. Pour notre projet nous avons choisi d'utiliser un système de message basé sur un id accolé au contenu du message. En voici un exemple : 100{50000}. Cet exemple est la requête utilisée pour se connecter, le premier nombre « 100 » est l'id du message et le nombre entre « { » est le numéro du port sur lequel le serveur répondra.

```
_socket.BeginReceiveFrom(state.buffer,  offset: 0, BufSize, SocketFlags.None, ref _epFrom, _recv = ar =>
{
    var so = (State) ar.AsyncState;
    int bytes = _socket.EndReceiveFrom(ar, ref _epFrom);
    _socket.BeginReceiveFrom(so.buffer,  offset: 0, BufSize, SocketFlags.None, ref _epFrom, _recv, so);

    var msg = Encoding.ASCII.GetString(so.buffer,  index: 0,  count: bytes);
    var from = _epFrom.ToString();
    /*
    Console.WriteLine("RECV: {0}: {1}, {2}", from, bytes,
        msg);
    Console.WriteLine();
    */
    ReadMessage(msg, out var id, out var content);



    switch (id)
```

Listing 16

La fonction **ReadMessage** permet de séparer le message de son id en utilisant une expression régulière afin de détecter l'id en début de message. Ensuite l'id est enlevé du message total et le résultat obtenu sera écrit dans la variable content. L'id, quant à lui, est écrit dans la variable du même nom. Sur le listing 17 vous trouverez la fonction **ReadMessage** ainsi que la fonction **ReadIp,** qui fonctionne de façon similaire à la première, et permet de récupérer l'adresse ip du client qui envoie une requête de connexion.

```
private static string ReadIp(string from)
{
    var rx = new Regex( pattern: @"[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}",
        options: RegexOptions.Compiled | RegexOptions.IgnoreCase);
    return rx.Match(from).ToString();
}
1 usage  CDulouard
private static void ReadMessage(string message, out int id, out string content)
{
    var rx = new Regex( pattern: @"^[0-9]*",
        options: RegexOptions.Compiled | RegexOptions.IgnoreCase);
    if (int.TryParse(rx.Match(message).ToString(), out var x))
    {
        id = x;
        var idLength = id.ToString().Length;
        content = message.Substring( startIndex: idLength,  length: message.Length - idLength);
        return;
    }

    content = message;
    id = -1;
}
```

Listing 17

Comme vous pouvez le voir sur le listing 18, lors de la réception d'un message de connexion, la fonction stoppe le précèdent socket, si besoin, puis lit l'adresse ip de l'émetteur du message et démarre un nouveau **UdpSocket** qui permettra d'envoyer des messages vers l'adresse ip de l'émetteur du message et sur le port spécifié. Enfin on utilise la fonction **Send** afin d'envoyer un message de confirmation de connexion. Nous tenons à préciser que la connexion n'est pas sécurisée, il n'y a pas de chiffrement ni de protections afin d'empêcher un client non désiré de se connecter. Cela est volontaire car un tel dispositif aurait compliqué encore plus la communication et aurait augmenté le temps de latence. De plus cela n'est pas nécessaire au bon fonctionnement du projet.

```
case 100:
    var rx = new Regex( pattern: @"[0-9]*",
        options: RegexOptions.Compiled | RegexOptions.IgnoreCase);

    var port = content.Substring( startIndex: 1, length: content.Length - 1);

    if (int.TryParse(rx.Match(port).ToString(), out var x))
    {
        if (_client != null)
        {
            _client.Stop();
        }

        _clientPort = x;
        _clientIp = ReadIp(from);

        //Console.WriteLine(_clientIp+ ":" + _clientPort);

        _client = new UdpSocket();
        _client.Client(_clientIp, _clientPort);
        _client.Send( text: "Connected");
        //Console.WriteLine(_clientIp+ ":" + _clientPort);

    }
    else
    {
        _clientPort = 0;
    }
    break;
```

Listing 18

La fonction **Send** se contente de convertir le message en byte et de l'envoyer à l'aide du socket client.

```
public void Send(string text)
{
    byte[] data = Encoding.ASCII.GetBytes(text);
    _socket.BeginSend(data, offset: 0, size: data.Length, SocketFlags.None, callback: (ar) =>
    {
        var so = (State) ar.AsyncState;
        int bytes = _socket.EndSend(ar);
        //Console.WriteLine("SEND: {0}, {1}", bytes, text);
    }, state);
}
```

Listing 19

Intéressons-nous à présent aux quatre autres commandes qu'un client peut envoyer au serveur. La première commande a pour id 1000 et permet d'envoyer de nouvelles consignes au robot afin qu'il bouge aux positions requises.

Le contenu du message envoyé par le client distant doit être selon ce format :

```
{
        "legFrontLeftBot": 10.5,
        "legFrontLeftTop": 10.5,
        "shoulderFrontLeft": 10.5,
        "legFrontRightBot": 10.5,
        "legFrontRightTop": 10.5,
        "shoulderFrontRight": 10.5,
        "legBackLeftBot": 10.5,
        "legBackLeftTop": 10.5,
        "shoulderBackLeft": 10.5,
        "legBackRightBot": 10.5,
        "legBackRightTop": 10.5,
        "shoulderBackRight": 10.5
}
```

Il ‘agit d’un objet Json qui sera compris par le simulateur et qui assignera les positions envoyées à chaque moteur.
Après la lecture de l’objet Json, le serveur renvoie le statut actuel du robot sous la forme d’un objet Json comme suit :


```
{
    "_motorsDatas": {
        "LegFrontLeftBot": 9.994821,
        "LegFrontLeftTop": 10.15787,
        "ShoulderFrontLeft": -10.20117,
        "LegFrontRightBot": 10.05643,
        "LegFrontRightTop": 10.03171,
        "ShoulderFrontRight": 10.14973,
        "LegBackLeftBot": 10.05437,
        "LegBackLeftTop": 10.14999,
        "ShoulderBackLeft": -10.2139,
        "LegBackRightBot": 10.13692,
        "LegBackRightTop": 10.18322,
        "ShoulderBackRight": 10.09452
    },
    "_lidarDatas": [
        [-45, 0 ,0],
        [-25, 0 ,261,5513],
        [-5, 0 ,0],
        [15, 0 ,539,2242],
        [35, 0 ,0]\n    ],
    "_sensorDatas": {
        "roll": -0.1235334,
        "pitch": -1.524489,
```


```
        "yaw": -10.94584,
        "posX": -7.599296,
        "posY": 39.08562,
        "posZ": -1183.297\n    },
"score": -10,
"timeOut": 1,
"isDown": 0,
"isColliding": 0,
"finished": 0
}
```


On y retrouve les données des moteurs, du lidar (plus le nombre de rayon est grand plus le nombre de points renvoyés est grand), l'attitude du robot et sa position. On y retrouve aussi différents flags qui serviront pour l'apprentissage de la marche mais nous reviendrons plus en détails sur cette partie dans la section dédiée à cela.

```
case 1000:
    /*    received new target position    */
    if (ReadIp(from) == _clientIp && _clientPort != 0)
    {
        ReadTargetPositions(content);
        _client.Send(Manager.status);
    }
    break;
```

Le listing 20 représente le cas dont nous avons parlé. Tous les cas suivants utiliseront la même structure à savoir lecture du message, exécution d'éventuelles fonctions et envoi d'une réponse.

La requête suivante a pour id 1001, elle ne renvoie que le statut actuel du robot comme nous l'avons vu pour la requête d'id 1000. Cela sert notamment à connaitre l'état du robot lors du début d'un épisode d'apprentissage.

La requête d'id 1002 permet de recharger la scène afin de repartir à l'état initial. Nous ne détaillerons pas le code de la fonction de **respawn** car elle fonctionne exactement comme les fonctions utilisées pour changer de scène que nous avons vues lorsque l'on parlait des boutons (on se contente de recharger la même scène mais sans éteindre le serveur).

La requête d'id 1003 permet de recevoir uniquement le score. Nous ne l'utilisons pas mais elle reste disponible au besoin.

L'ajout d'une nouvelle requête est très simple, il suffit de rajouter un cas au switch case et d'exécuter les fonctions adaptées.

Fonctionnement des outils d’apprentissage

Nous allons aborder dans cette section la dernière fonctionnalité du simulateur dont nous n’avons pas parlé jusqu’ici à savoir les outils mis à disposition pour l’apprentissage.
En plus des commandes serveur que le client peut appeler lorsqu’il le souhaite, le simulateur contient plusieurs outils pour aider à l’apprentissage automatique. Ces outils incluent plusieurs flags qui renseignent divers évènements importants pour l’ia comme une chute ou une collision. Il y a aussi un score propre au simulateur qui permet de retourner une récompense basique pour l’intelligence artificielle qui est connectée. Ces informations sont transmises à la fin de l’objet Json de statut transmis par le serveur lors de la réception d’une requête adaptée. Vous pouvez revoir ci-dessous les variables concernées :

"score": -10,
"timeOut": 1,
"isDown": 0,
"isColliding": 0,
"finished": 0

Intéressons-nous d’abord au fonctionnement des flags.

Chaque Scène contenant le robot possède plusieurs informations dont un temps d’épisode maximum.

```
private Vector3 _spawnPoint;
private Vector3 _finishPoint;
private int _gameDuration;
```

Sur le listing 21, on peut voir les trois informations disponibles pour chaque terrain.

Dans le script **Controller** plusieurs coroutines se chargent de vérifier régulièrement l’état du la scène afin de mettre à jour les flags.

Le flag **timeOut** est contrôlé par la coroutine **Timer** que vous trouverez sur le listing 22 :

```
private IEnumerator Timer(int time)
{
    var count = time;
    while (count != 0)
    {
        yield return new WaitForSeconds( seconds: 1);
        count -= 1;
    }
    _timeOut = 1;
}
```

Listing 22

Il s'agit d'un simple décompte qui s'actualise toutes les secondes. Si le décompte atteint 0, le flag timeOut est mis à 1.

Le flag **isDown** permet de savoir si le robot est tombé (de façon grossière). Voici le code de la coroutine associée (Listing 23) :

```
private IEnumerator CheckIfDown()
{
    yield return new WaitForSeconds( seconds: 0.1f);
    while (true)
    {

        var datas = Manager.datas.GetSensorDatas().ToDictionary(x => x.Key, x => x.Value);
        if (datas["roll"] > 90f || datas["roll"] < -90f || datas["pitch"] > 90f || datas["pitch"] < -90f )
        {
            _isDown = 1;
        }
        else
        {
            _isDown = 0;
        }
        yield return new WaitForSeconds( seconds: 0.1f);
    }
}
```

Listing 23

Toutes les 0.1 secondes, la coroutine regarde si les angles sur les axes x et z sont supérieurs à 90 degrés ou inferieurs à -90 degrés (à partir de ces valeurs d'angle on considère que le robot a de très fortes chances de tomber et de se coincer sur le dos). Si cette condition est remplie le flag **isDown** est mis à l'état 1, sinon il est mis à l'état 0.

Le flag **IsColliding** permet de savoir si le robot a heurté un objet.

```
public class CollisionManager : MonoBehaviour
{
    Event function  CDulouard
    private void OnCollisionStay(Collision collisionInfo)
    {
        if (collisionInfo.collider.name != "Ground")
        {
            Controller.SetIsColliding( val: 1);
        }
    }
    Event function  CDulouard
    private void OnCollisionExit(Collision collisionInfo)
    {
        if (collisionInfo.collider.name != "Ground")
        {
            Controller.SetIsColliding( val: 0);
        }
    }

}
```

Sur chaque pièce du robot est attaché le script suivant (Listing 24) :

Ce Script est déclenché si la pièce rentre en collision avec un autre objet. Le script vérifie que la collision n'est pas causée par le sol et si ce n'est pas le cas met le flag à 1 (il le fera en continu tant que la pièce est en collision).

Lorsque la pièce n'est plus en collision, on remet le flag à 0 (cela ne se produit qu'une fois lors de la sortie de collision).

Ainsi grâce à ce script, on peut savoir en permanence si le robot est en collision ou non.

Le flag **finished** et le score sont calculés dans la coroutine **CheckForProgress** dont voici le code (Listing 25) :

```
private IEnumerator CheckForProgress()
{
    /*    Refresh the robot's score    */
    var finish = _currentMap.GetFinishPoint();
    var prevDistance = Vector3.Distance( a: generalPurposeSensor.transform.position,  b: finish);
    while (true)
    {
        var dFromFinish = Vector3.Distance( a: generalPurposeSensor.transform.position, b: finish);
        _score = (int)(prevDistance - dFromFinish - 2);
        if (_isDown == 1)
        {
            _score = -10;
        }
        if (_isColliding == 1)
        {
            _score = -10;
        }
        if (_timeOut == 1)
        {
            _score = -10;
        }
        if (dFromFinish <= 30)
        {
            _score = 100;
            _finished = 1;
        }
        prevDistance = dFromFinish;
        yield return new WaitForSeconds( seconds: 0.1f);
    }
}
```

Listing 25

Cette fonction va dans un premier temps vérifier si le robot a avancé depuis le dernier appel de celle-ci. Pour cela, on calcule la distance entre le point d'arrivée (qui est stocké dans les données du terrain que nous avons vues précédemment) et la position actuelle. Ensuite, on fait la différence entre ce résultat et la position précédente. On enlève aussi 2 au résultat obtenu afin d'obtenir un premier score qui reflètera l'avancée du robot mais qui le pénalisera s'il reste sur place.
Ensuite on vérifie chaque flag en fonction de l'ordre de priorité de ces derniers. Si une condition est satisfaite, le score associé précédemment sera remplacé par la nouvelle valeur.
La dernière condition est la condition la plus importante, il s'agit de la condition de fin. Si elle est vraie, on renvoie la plus haute valeur de score et on met le flag **finished** à 1 pour signifier à l'ia que son travail est fini.
L'utilisation de toutes ces informations est facultative pour l'ia, elle peut calculer si elle le souhaite un score différent à partir de toutes les données qu'on lui renvoie. De

plus la variation des flags ne change rien à la simulation. Par exemple si l'IA souhaite recommencer la partie, lorsqu'elle tombe ou qu'elle finit la partie, c'est à elle de dire à la simulation de recommencer. Si elle ne le fait pas le simulateur continuera de simuler de façon normale.

Apprentissage d'un robot grâce à la simulation

Important :

La lecture de cette partie nécessite des connaissances solides en python, mathématiques et en machine Learning. Bien que nous ayons essayé de vulgariser au maximum notre travail, la compréhension de ce dernier pourrait être fortement diminuée si vous ne maîtrisez pas les bases des trois concepts précédemment mentionnés.

Pour un souci de lisibilité, et, afin de rendre le travail plus compréhensible, seules certaines portions du code ont été affichées dans cette partie.

Vous trouverez le code en question sur notre GitHub à l'adresse suivante :

https://github.com/Remydeme/PROROK

Nous utilisons Python 3.7 ainsi que les librairies suivantes :

- gym
- tensorflow 2.0.0-alpha0
- tensorflow-gpu 2.0.0-alpha0
- roboschool

Apprentissage par renforcement

L’entrainement d’une intelligence artificielle dans un robot hors simulation comporte des risques élevés de dégâts matériels. Il est donc préférable d’entraîner un agent dans un environnement simulé semblable à la réalité, et, une fois celui-ci suffisamment performant, il nous est alors possible de le tester dans un contexte réel. Le simulateur APOLLO a été construit dans le but d’entraîner nos différents modèles d’intelligence artificielle. Durant notre projet nous avons utilisé des algorithmes d’une branche du machine learning nommée l’apprentissage par renforcement.

L'apprentissage par renforcement est l'un des domaines du machine learning le plus passionnant. Il est devenu célèbre notamment grâce à **Deep-Mind** en 2013 avec leur IA alpha-go qui a battu le champion du monde de GO. Il consiste, pour un agent autonome, à apprendre les actions à opérer, à partir d'expériences, de façon à optimiser une récompense au cours du temps. L'agent évolue dans un environnement qui lui est inconnu et prend ses décisions en fonction de son état courant. En retour, l'environnement procure à l'agent une récompense, qui peut être positive ou négative. L'intelligence artificielle cherche, au travers d'expériences itérées, un comportement décisionnel appelé stratégie ou **politique** qui est une fonction associant à l'état courant l'action optimale à exécuter, afin qu'elle maximise la somme des récompenses au cours du temps. Avant de vous présenter le modèle d'intelligence artificielle utilisé, il est important que nous définissions les termes clés de l'apprentissage par renforcement.

Quality-learning (Q-learning).

Un processus stochastique sans mémoire ou chaîne de Markov possède un nombre fini d’états et évolue aléatoirement d’un état à l’autre à chaque étape. Le passage d’un état S à un état S' a une probabilité fixée P qui dépend uniquement du couple (S, S’) et non des états précédents (processus sans mémoire). Dans les années 50, Richard Bellman a étudié les processus de décision markoviens. Ils sont semblables aux chaînes de Markov. À chaque étape, un agent peut choisir entre plusieurs actions possibles et les probabilités des transitions dépendent de l'action choisie. De plus, certaines transitions entre états renvoient une récompense négative ou positive. L'objectif de l'agent est de trouver la politique qui maximise les récompenses au fil du temps.

À la suite de son étude, Bellman a trouvé une méthode pour estimer la valeur optimale de tous les états S.

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

Figure a

L'équation ci-dessus est l'équation d'optimalité de Bellman (Figure a).

A quelles questions répond cette équation ?

- Sachant que l'on suit une politique consistant à maximiser la récompense future et que l'on est à l'état S, quelle est l'action A que l'on peut faire qui maximise la récompense future de notre agent ?

- Quelle est la valeur de l'état S sachant que nous effectuons l'action A ?

En effet, si un agent suit une politique optimale alors cette équation nous dit que la valeur de l'état S est la somme de récompenses futures qu'il peut espérer (P(A, A', S)) en faisant des actions optimales multipliées par le taux de rabais gamma à laquelle s'ajoute la récompense obtenue à l'état S. C'est une façon d'estimer la valeur optimale de tous les états. Connaitre la valeur optimale d'un état est important, car cela nous permet d'évaluer une politique.

Bellman a également trouvé un algorithme permettant de déterminer la valeur du couple (S, A) état-action.

L'algorithme d'apprentissage temporel différentiel.

Monte Carlo $$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

TD Learning $$V(S_t) \leftarrow \underbrace{V(S_t)}_{\text{Previous estimate}} + \alpha[\underbrace{R_{t+1}}_{\text{Reward t+1}} + \underbrace{\gamma V(S_{t+1})}_{\text{Discounted value on the next step}} - V(S_t)]$$

TD Target

Figure b

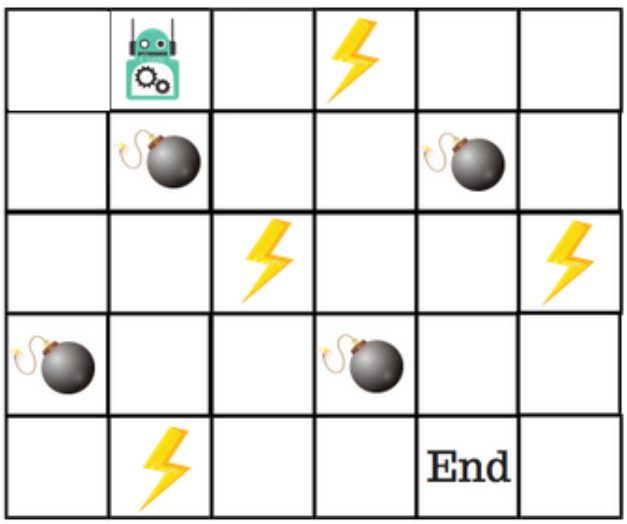
- alpha : taux d'apprentissage. Il nous permet de définir à quel point on accepte la nouvelle valeur de l'état. Il est compris entre 0 et 1 et est, en général, très petit (de l'ordre de 1e-3). Comme pour la descente de gradient, il permet de gérer la convergence de notre algorithme, mais à la différence de la descente de gradient, alpha doit être augmenté à mesure que notre agent apprend afin que l'algorithme converge.

- gamma : taux de rabais nous permettant de préciser l'importance des actions futures. Une valeur de 1 pour gamma signifie que les actions futures ont autant d'importance que les actions présentes. Une valeur proche de zéro signifierait le contraire.

Chaque état S conserve une moyenne mobile exponentielle des récompenses immédiates à laquelle on ajoute les récompenses qu'il obtiendra dans le futur en suivant la politique qui est de faire des actions maximisant les récompenses futures.

Q-Table

Avec l'équation différentielle temporelle, il nous est possible de concevoir une matrice contenant la valeur pour chaque action possible et pour chaque état de l'environnement, c'est une Q-Table.



Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

Figure c

À partir de cette table il est alors possible pour notre agent d'évoluer dans l'environnement et de maximiser ses récompenses lorsque celui-ci suit une politique optimale. Vous pourrez trouver une implémentation de l'algorithme de Monte-Carlo utilisant l'algorithme d'apprentissage temporel différentiel sur notre github.

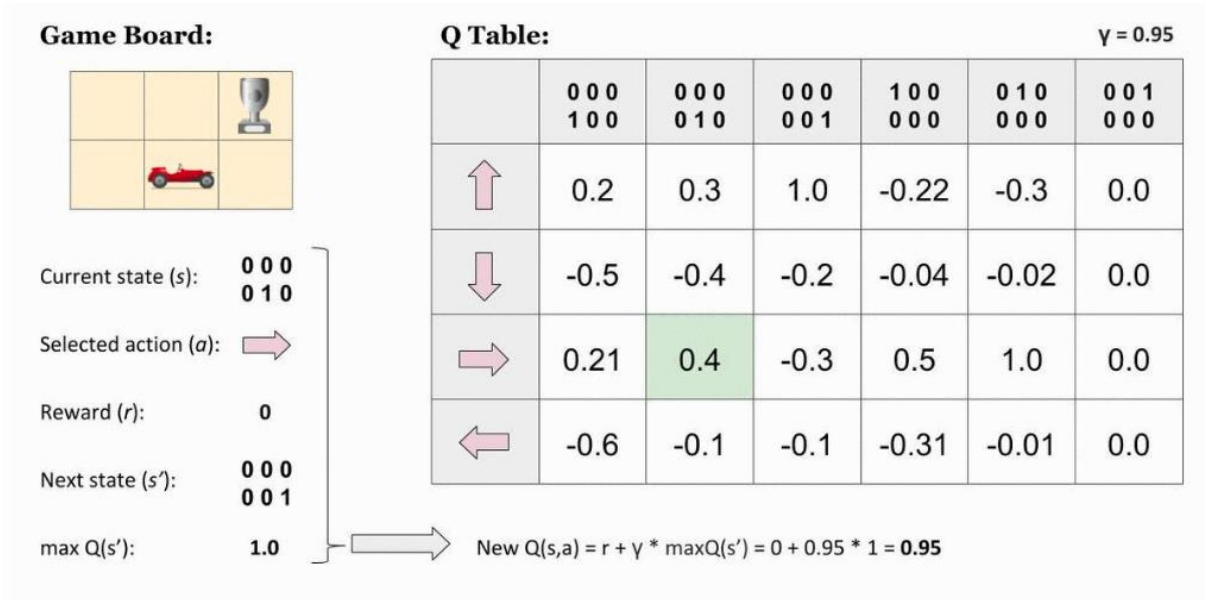


Figure d

Deep Q-learning

L'objectif de notre projet est de concevoir un agent permettant d'évoluer dans un environnement constitué d'un nombre d'états finis. Bien que fini, cet environnement est composé d'un grand nombre d'états. Considérons que notre agent souhaite apprendre à jouer à Pac-Man. Il y a 256 pastilles dans l'environnement. À tout instant, ces pastilles peuvent avoir été mangées ce qui nous donne 2^256 états possibles uniquement en prenant en compte les pastilles. L'apprentissage Q n'est pas adapté aux moyens et grands environnements.

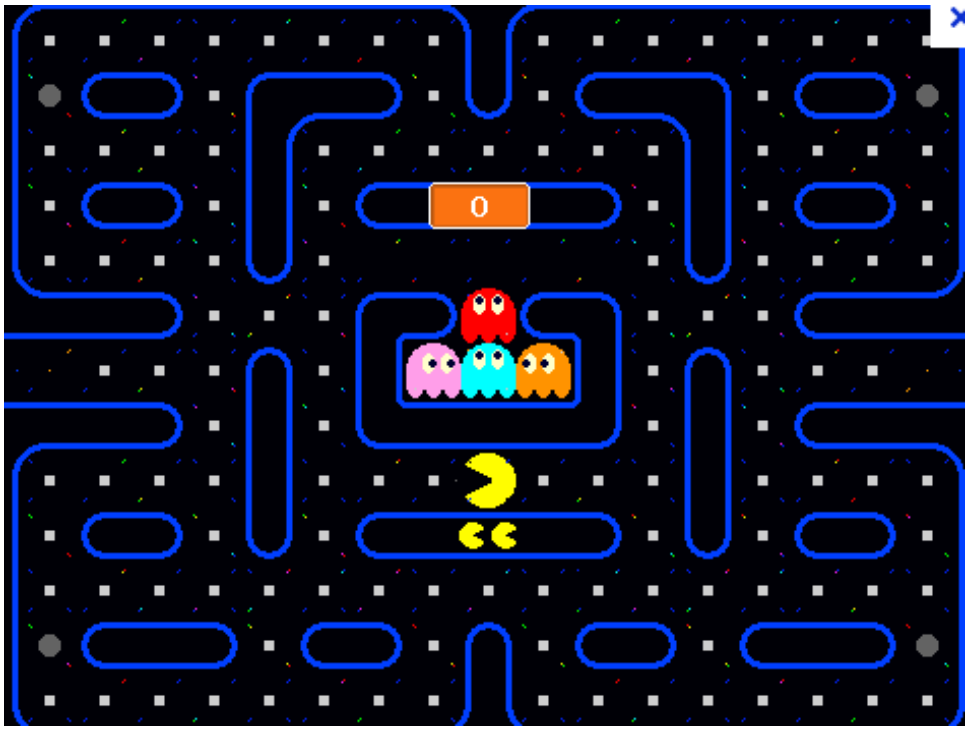


Figure e

La solution à ce problème a été proposée par Deep-Mind. Nous devons approximer notre Q-function en utilisant un réseau de neurones. C'est l'apprentissage par renforcement profond.

Comment approximer notre Q-function ?

Grâce à l'équation d'optimalité de Bellman, nous savons que notre réseau de neurones, si nous lui donnons pour valeur d'entrée notre action A ainsi que notre état S, doit nous fournir en sortie la valeur de récompense de l'état plus la valeur de la récompense de l'état suivant S', sachant que l'on a fait l'action optimale A' multiplier par le taux de rabais gamma. Le réseau de neurones d'apprentissage profond (DQN) prédit la valeur V (A, S) en utilisant A et S comme entrée et nous utilisons S' et A' pour calculer V(S', A').

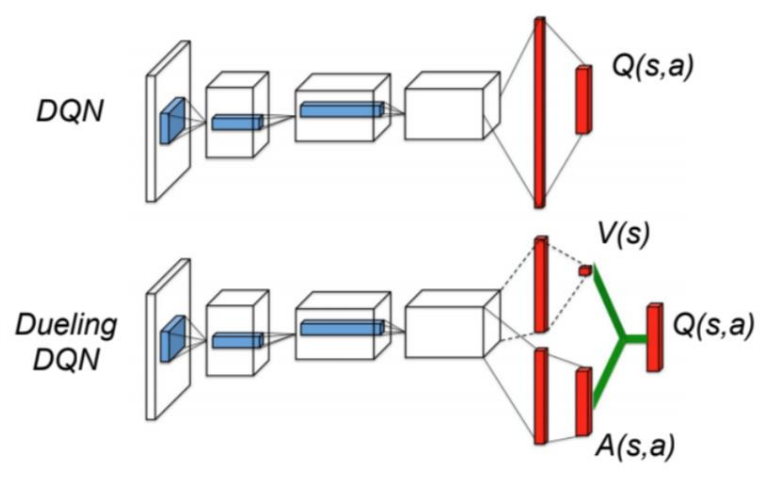


Figure f

Avec V(S, A) et V(S', A'') il est alors possible d'entraîner notre réseau de neurones. En calculant l'erreur V(S,A) – Cible, nous pouvons calculer le gradient delta qui nous permettra de modifier les poids de notre réseau de neurones durant la descente de gradient afin de diminuer l'erreur de notre Q-function. Après entraînement, notre réseau de neurones est l'approximation de notre Q-function. Il est capable de calculer la valeur du couple (S, A) de notre environnement avec une certaine précision qui dépend de son temps d'entraînement et des expériences qu'il a utilisées lors de l'apprentissage.

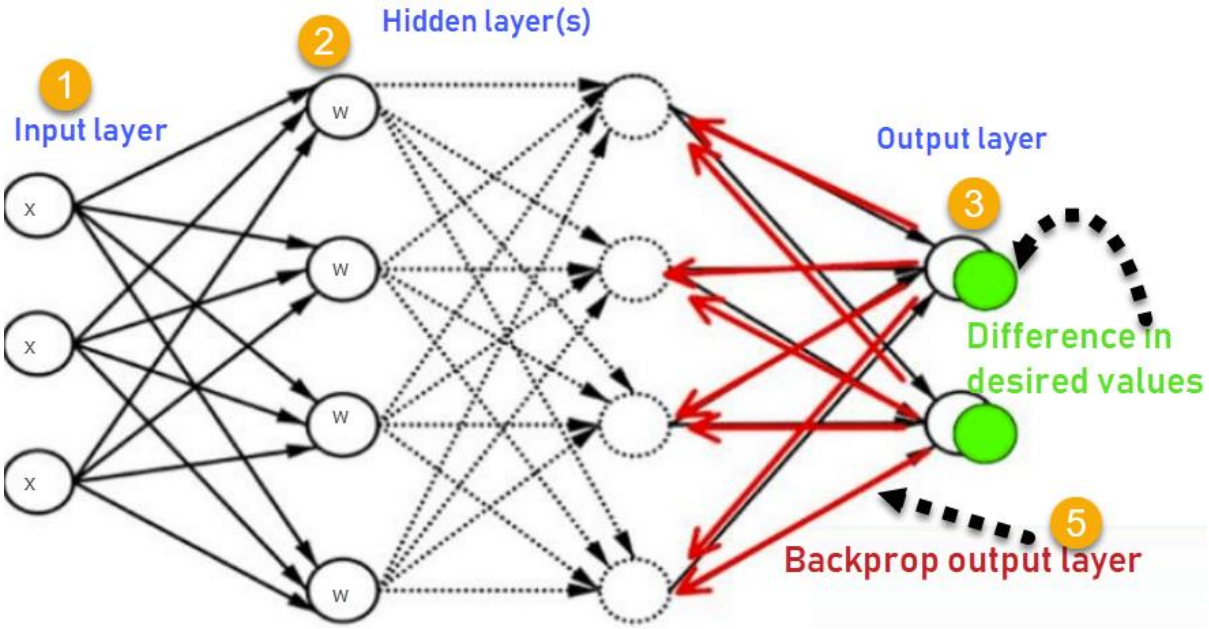


Figure g

Deep deteriministic policy gradient

Un robot est composé de plusieurs moteurs qui lui permettent de bouger ses différents membres. Il est également composé de capteurs, lidar et ou caméras qui lui permettent de se positionner dans son environnement et également de détecter d'éventuels obstacles. Toutes ces informations constituent des données d'entrée pour notre réseau de neurones. Les moteurs présents dans nos futurs robots sont pilotés *via* des valeurs continues et non discrètes. Il nous a donc fallu construire un modèle capable de fournir en sortie des valeurs continues pour chacun des moteurs contrôlant une articulation de notre robot. **Le Deep Deterministic Policy Gradient (DDPG)** est un algorithme qui apprend parallèlement une Q-function et une politique. Elle utilise l'équation de Bellman afin d'approximer une Q-function, et utilise l'approximation de la Q-function afin d'apprendre la politique. C'est un modèle utilisant la méthode de "l'acteur-critique", fournissant uniquement en sortie des valeurs continues et c'est un algorithme d'apprentissage hors ligne ou off-policy.

Actor-Critic

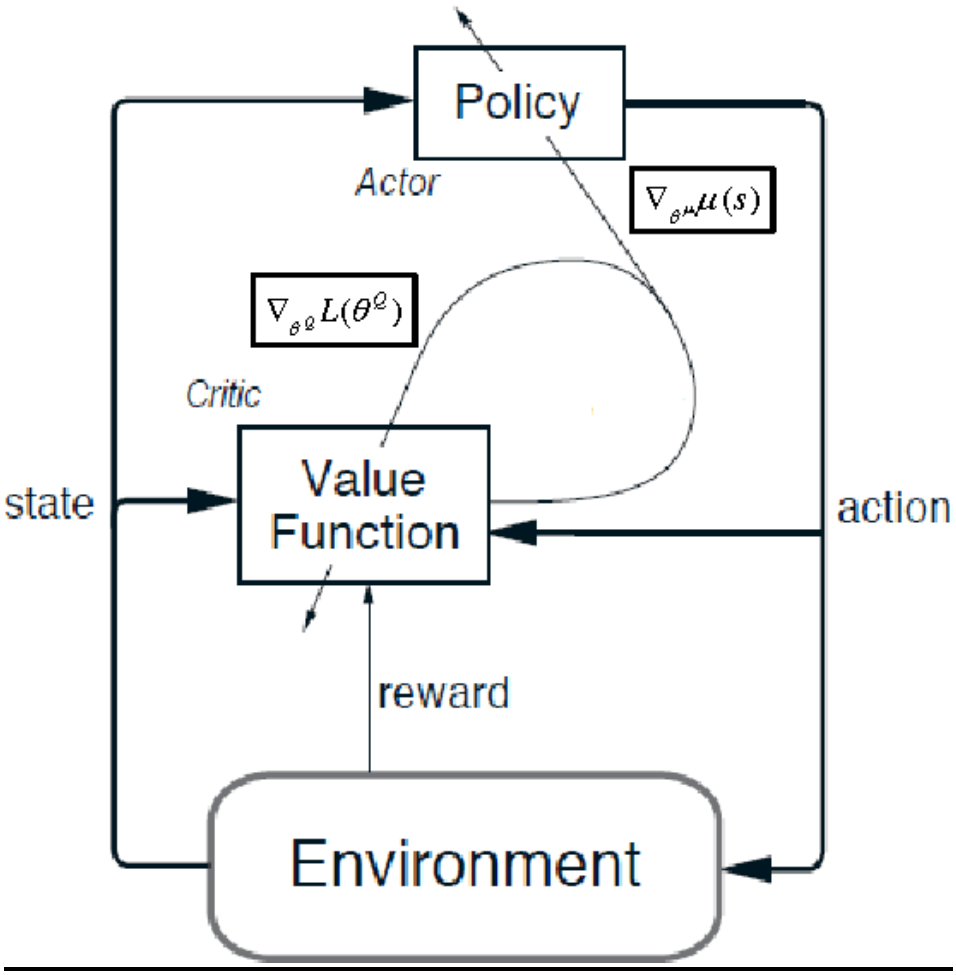


Figure h

Imaginez un acteur faisant un ensemble d'actions et un critique notant chacune de ces actions. L'acteur utilise les notes du critique afin de s'améliorer et le critique s'améliore à travers les expériences afin de mieux noter l'acteur. C'est cela la méthode de actor-critic.

Off policy

Le DDPG est l'algorithme d'apprentissage hors ligne. Il apprend à partir d'expériences stockées dans une mémoire de “rejeu”. L'algorithme est capable d'utiliser des expériences réalisées par un acteur plus ancien pour s'entraîner. Ceci est possible car l'équation de Bellman ne se soucie pas de comment l'expérience a été obtenue, des transitions (S, A, S') ou de comment elles ont été sélectionnées. Notre critique doit satisfaire l'équation de Bellman et doit pouvoir calculer la valeur de tous les couples (S, A) possibles.

Algorithme du DDPG

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure i

L'algorithme du DDPG utilise quatre réseaux de neurones. Un critique, un acteur et leurs cibles.

$$\text{Parameters:}$$

$$\theta^{Q} : \text{Q network}$$

$$\theta^{\mu} : \text{Deterministic policy function}$$

$$\theta^{Q'} : \text{target Q network}$$

$$\theta^{\mu'} : \text{target policy network}$$

Figure j

Durant la phase d'entraînement, l'agent joue des parties. Lorsque l'aent fait une action dans la partie, nous stockons dans une mémoire d'entraînement la récompense (R) obtenue, l'état futur (S') dans lequel nous nous retrouvons, le couple (A, S) et également si la partie s'est terminée suite à cette action (D). À la suite de chaque action nous entraînons notre agent.

- Nous sélectionnons un échantillon aléatoire d'éléments de taille N (cela réduit la corrélation entre les expériences dans le lot d'entrainement.).

- A l'aide de l'équation temporelle de Bellman, nous entrainons notre critique. Nous utilisons (S, A) de notre échantillon pour calculer V(S, A), nous utilisons l'état futur stocké pour calculer A'. Et A' est utilisé pour pouvoir calculer la valeur cible Y(S', A').

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) \max_{a'} Q_\phi(s',a') \right) \right)^2 \right]$$

Figure k

Il est alors possible d'entrainer notre critique en utilisant la descente de gradient pour minimiser l'erreur entre Y et V(S,A).

L'agent va chercher à maximiser la récompense obtenue pour l'action qu'il a fait, soit maximiser V(S, A).

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s)) \right].$$

Figure l

Nous utilisons deux réseaux de neurones pour notre critique. Le premier apprend à évaluer notre couple état-action, le deuxième est utilisé pour calculer la valeur cible Y(A',S'). Il en est de même pour notre agent où le deuxième réseau de neurones sert à calculer la valeur de A'. **Deep-Mind** a montré que l'utilisation d'un réseau de neurones cible permettait d'améliorer considérablement les performances de l'algorithme. En effet, le fait que le réseau de neurones fixe lui-même sa cible et tente de l'atteindre rend l'algorithme instable. Ceci mène à l'apparition de boucle de rétroaction l'algorithme peut diverger, se geler et osciller.

Les paramètres du réseau cible sont mis à jour par soft update (Mise à jour légère) à chaque étape d'entraînement. C'est ce que l'on appelle la moyenne de Polyak. Au lieu de copier toutes les N étapes les paramètres du réseau principal dans la cible, on copie à chaque étape d'entraînement progressivement les paramètres du réseau principal dans la cible.

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

$$\text{where} \quad \tau \ll 1$$

Figure m

Les étapes de jeu et d'entraînement sont répétées jusqu'à ce que l'agent converge et obtienne un score suffisant.

Twin delayed deep deterministic policy gradient

Nous avons implémenté une optimisation du DDPG. Le TD3 ou Twin delayed deep deterministic policy gradient utilise trois astuces pour améliorer les performances de l'agent :

1. Deux critiques :

    -On utilise deux critiques pour entrainer notre agent.
    -On choisit la valeur la plus petite prédite par nos critiques pour calculer la valeur cible.
    -On minimise la somme des erreurs des deux critiques pour les améliorer. En faisant cela on réduit l'erreur et la variance de nos modèles.

2. La mise jour de la politique (agent) se fait de façon décalée par rapport au critique. En faisant cela, on améliore notre politique car la variance du critique est réduite. On a plus de chance en faisant cela de mettre à jour notre agent avec une critique qui s'est améliorée.

3. On ajoute un bruit normal à nos actions afin d'encourager notre modèle à l'exploration N(0, sigma).

Code

Nous avons utilisé le framework open source tensorflow (version 2.0) pour concevoir notre modèle. Tenorflow est un outil très puissant qui permet de concevoir des modèles d'apprentissages automatiques. Il utilise le principe des graphes. Toutes nos opérations sont représentées dans un graphe qui est décomposé en plusieurs morceaux, et, celles-ci sont exécutées en parallèle. Ainsi, il nous est possible d'entraîner des modèles d'apprentissages automatiques avec des données gigantesques en utilisant la puissance de cartes graphiques.

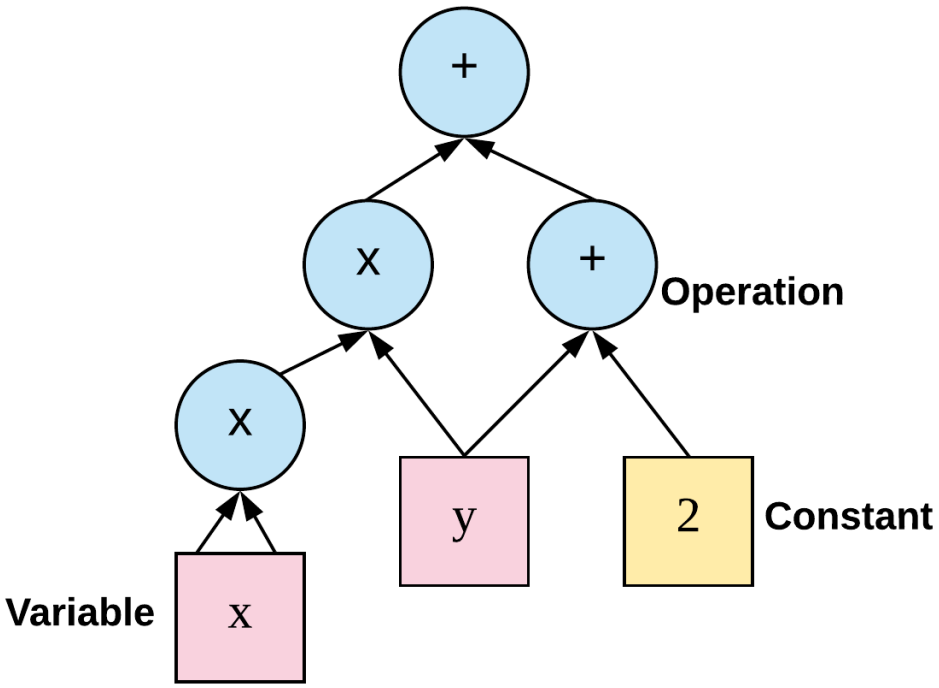


Figure n

- *(XX)Y + (y + 2)*

Graphes des réseaux neuronaux

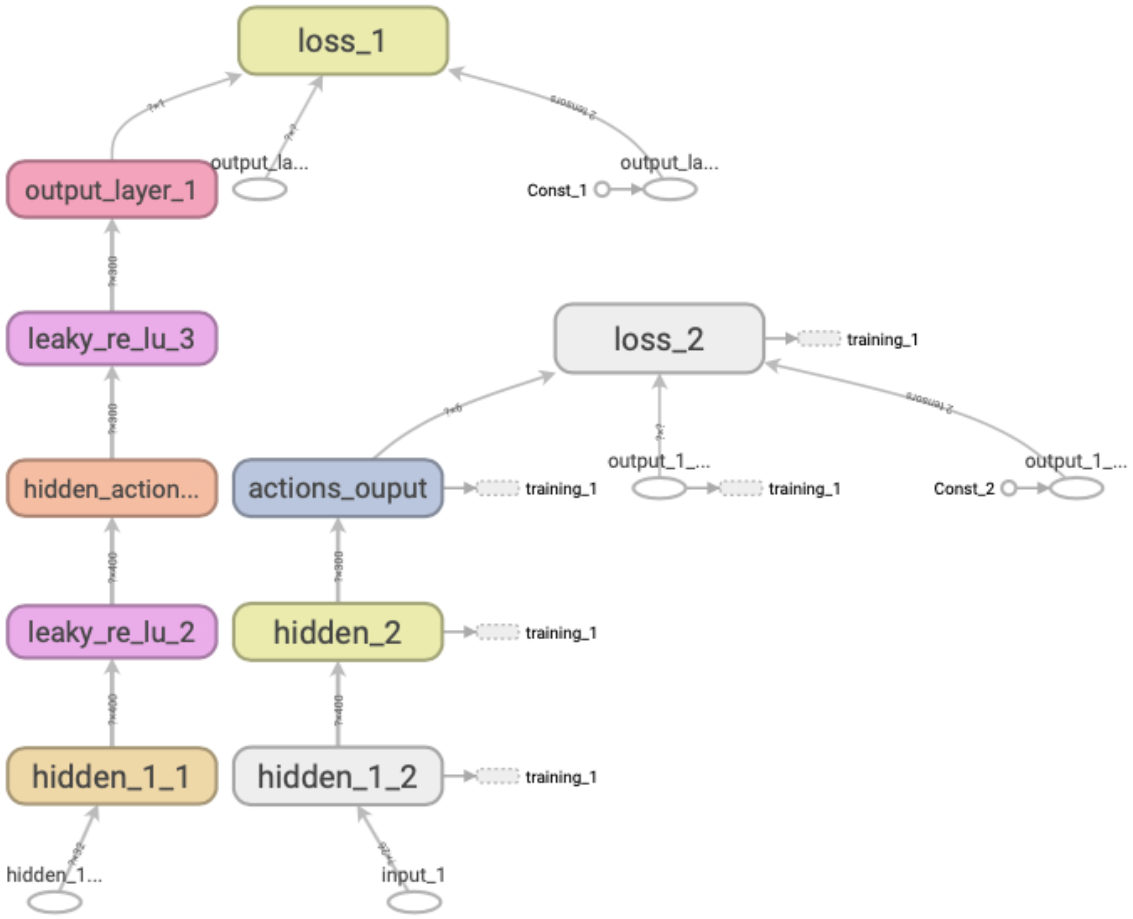


Figure o

Classe Critique

```python
class Critics:

    hidden_1 = 400
    hidden_2 = 300

    def __init__(self, env, lr=1e-3):
        input_dim = env.observation_space.shape
        action_dim = env.action_space.shape
        self.__lr = lr
        self.__initializer = k.initializers.he_uniform(seed=0)
        self.optimizer = k.optimizers.Adam(learning_rate=lr)
        self.valueNet = self.buildModel(input_dim=input_dim, action_dim=action_dim)
        self.targetValueNet = self.buildModel(input_dim=input_dim, action_dim=action_dim)
        self.valueNet.set_weights(self.targetValueNet.get_weights())


    def buildModel(self, input_dim, action_dim):

        model = k.Sequential()

        model.add(k.layers.Dense(self.hidden_1, input_dim=(input_dim[0] + action_dim[0]), activation='relu', name='hidden_1'))

        model.add(k.layers.Dense(self.hidden_2, activation='relu', name='hidden_action'))

        model.add(k.layers.Dense(1, activation='linear', name='output_layer'))

        return model

    def computeValue(self, state, actions):
        state_and_action = tf.concat([state, actions], axis=1)
        value = self.valueNet(state_and_action)
        return value

    def computeTargetValue(self, state, actions):
        state_and_action = tf.concat([state, actions], axis=1)
        value = self.targetValueNet(state_and_action)
        return value

    def softCopy(self, tau=0.005):
        target_pars = self.targetValueNet.get_weights()
        value_pars = self.valueNet.get_weights()
        index = 0
        for target_par, value_par in zip(target_pars, value_pars):
            target_par = target_par * (1 - tau) + value_par * tau
            target_pars[index] = target_par
            index += 1
        self.targetValueNet.set_weights(target_pars)

    def computeLosses(self, value, target):
        critic_loss = k.losses.mean_squared_error(target, value)
        return critic_loss
```

Listing a

Modèle DFF :

Notre critique est composé de deux réseaux de neurones profonds (Deep feed forward DFF). La taille d'entrée dépend de l'environnement étudié et du nombre d'actions possibles pour notre robot. Elle est égale à leur somme. Nous avons deux couches profondes. La première est composée de 400 neurones, la deuxième de 300 neurones. La critique fournie en sortie la valeur du couple (S, A) sa dimension est de 1.

Toutes les couches cachées utilisent la fonction d'activation « relu ». La couche de sortie n'utilise pas de fonction d'activation, elle est donc linéaire.

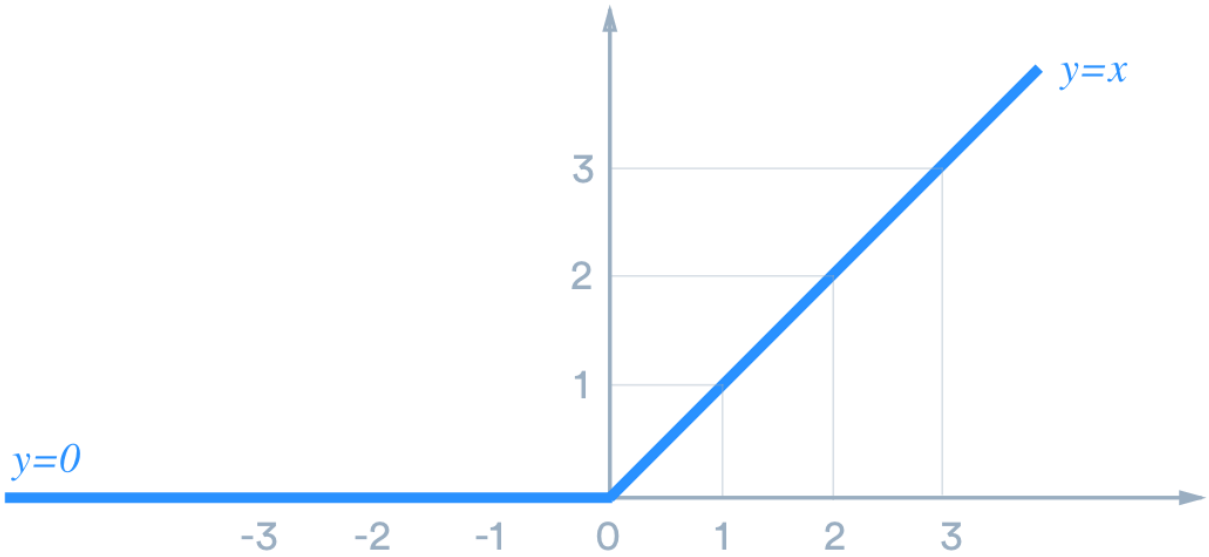


Figure p

Méthodes :

build_model : Cette méthode permet de construit notre modèle.
compute_value : Calcule la valeur de V(S, A) avec le réseau de neurones.
soft_copy: Elle réalise la soft copy.
computeLosses : Calcule l'erreur quadratique entre la cible et la prédiction faite par le critique principal.

Classe Policy

```python
class Model(k.Model):

    hidden_1_size = 400
    hidden_2_size = 300

    def __init__(self, input_dim, action_dim, action_high):
        super().__init__()
        self.action_high = action_high
        self.h1 = k.layers.Dense(self.hidden_1_size, activation='relu', name='hidden_1')
        self.h2 = k.layers.Dense(self.hidden_2_size, activation='relu', name='hidden_2')
        self.outputs = k.layers.Dense(action_dim, activation='tanh', name='actions_ouput')


    def call(self, inputs):
        x = self.h1(inputs)
        x = self.h2(x)
        x = (self.action_high[0] * self.outputs(x))
        return x



class Policy:



    def __init__(self, input_dim, action_dim, action_low, action_high, lr=1e-3):
        self.action_low = action_low
        self.action_high = action_high
        self.action_dim = action_dim
        self.policyNet = Model(input_dim=input_dim, action_dim=action_dim, action_high=action_high)
        self.policyTargetNet = Model(input_dim=input_dim, action_dim=action_dim, action_high=action_high)
        self.policyTargetNet.set_weights(self.policyNet.get_weights())

    def get_action(self, state):
        actions = self.policyNet(state)
        return actions[0]

    def evaluate_state(self, state):
        actions = self.policyNet(state)
        return actions


    def get_target_action(self, state):
        actions = self.policyTargetNet(state)
        return actions

    def softCopy(self, tau=0.005):
        target_pars = self.policyTargetNet.get_weights()
        value_pars = self.policyNet.get_weights()
        index = 0
        for target_par, value_par in zip(target_pars, value_pars):
            target_par = target_par * (1 - tau) + value_par * tau
            target_pars[index] = target_par
            index += 1
        self.policyTargetNet.set_weights(target_pars)
```

Listing b

Modèle DFF :

Notre classe est composée de réseaux de neurones profonds. Chaque réseau de neurones possède deux couches cachées : la première de 400 neurones et la deuxième 300 neurones. La taille de la sortie correspond au nombre d'actions à effectuer. Les fonctions d'activations des couches cachées sont des fonctions « relu ». La sortie utilise une fonction tangente hyperbolique. Cette fonction nous permet d'avoir une valeur de sortie comprise entre [-1, 1] essentielle au pilotage des moteurs du robot.

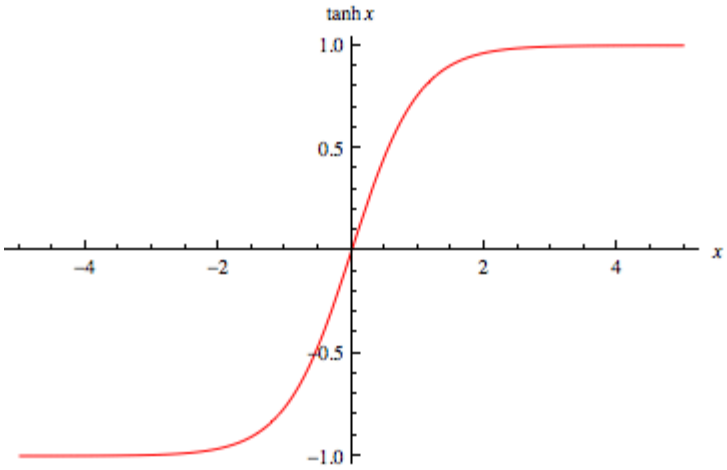


Figure q

Méthodes :
get_action(state) : Calcule avec le réseau de neurones principal la valeur de l'action pour l'état S passé en paramètre.
evaluate_action(state) : Cette méthode retourne un tenseur d'actions.
get_target_action : Retourne la valeur de l'action calculée par le NN cible.

Class ReplayBuffer

Cet objet, grâce à sa fonction sample, nous permet d'obtenir un échantillon aléatoire des expériences. C'est notre mémoire de rejeu.

```
"""

Replay buffer class

"""



class ReplayBuffer():
    def __init__(self, obs_dim, act_dim, size):
        self.obs1_buf = np.zeros([size, obs_dim], dtype=np.float32)
        self.obs2_buf = np.zeros([size, obs_dim], dtype=np.float32)
        self.acts_buf = np.zeros([size, act_dim], dtype=np.float32)
        self.rews_buf = np.zeros(size, dtype=np.float32)
        self.done_buf = np.zeros(size, dtype=np.float32)
        self.ptr, self.size, self.max_size = 0, 0, size

    def store(self, obs, act, rew, next_obs, done):
        self.obs1_buf[self.ptr] = obs
        self.obs2_buf[self.ptr] = next_obs
        self.acts_buf[self.ptr] = act
        self.rews_buf[self.ptr] = rew
        self.done_buf[self.ptr] = done
        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)


    def sample_batch(self, batch_size=32):
        index = np.random.randint(0, self.size, size=batch_size)
        return self.obs1_buf[index], self.obs2_buf[index], self.acts_buf[index], self.rews_buf[index], self.done_buf[index]
```
Listing c

Class AgentD3PGG

Cette classe est composée de :

- Deux Instance de Critics
- Une Instance d'Actor
- Une Instance de ReplayBuffer

Elle prend également en paramètres les hyper-paramètres de notre agent :

-env : objet contenant toutes les informations concernant l'environnement (dimension de l'action et des observations).
-actor_noise : le bruit ajouté aux actions de notre acteur principal N(0, actor_noise).
-target_noise : la valeur de la variance du bruit normal gaussien qui est ajouté aux actions calculées par notre cible N(0, target_noise).
- gamma : notre taux de rabais 0.99

Méthodes :

Update(batch_size, step) : c'est la fonction d'entraînement de notre agent. On prend un échantillon aléatoire de paramètres de la taille de batch_size. Nous devons convertir en tenseur ces tableaux contenant nos données d'entraînement.

On commence par déterminer les actions futures A' en utilisant notre acteur cible. Cette action A' que l'on a appelée next_action nous l'utilisons pour calculer notre target. Les target de nos deux critiques sont utilisées pour calculer deux Q(S', A'). Le plus petit est utilisé pour calculer la cible expected_value. Avec notre cible, il nous est possible de calculer l'erreur pour nos deux critiques. Une fois additionnés, nous calculons le gradient à l'aide de l'outil de différenciation de tensorflow GradientTape. Nos réseaux de neurones critiques sont optimisés par Adam qui utilise le gradient qu'on lui fournit en paramètre pour minimiser l'erreur en faisant une descente de gradient.

Un critique peu importe lequel des deux est choisi pour calculer la valeur des couples (S, A) de notre échantillon. Afin de maximiser cette valeur, nous calculons le gradient de -Q(S, A). Un optimizer est conçu pour minimiser l'erreur qui lui est passé en paramètre en lui donnant l'opposé de celle-ci il maximisera notre Q(S, A) . Que veut-on dire par cela ? Nous souhaitons que notre optimizer modifie la valeur de poids de notre réseau de neurones pour l'encourager à faire des actions qui vont dans la direction des récompenses données par notre critique. Un Q(S, A) élevé signifie que l'action a été bonne. Un Q(S, A) petit signifie le contraire. Le gradient ainsi calculé fera évoluer les réseaux neuronaux de notre acteur de cette façon.

Notre acteur est mis à jour deux fois moins souvent que nos critiques et la soft copy est réalisée avec la même fréquence.

```python
def update(self, batch_size=100, iterations=100):

  for step in range(iterations):

      state, next_state, actions, reward, done = self.replayBuffer.sample_batch(batch_size=batch_size)

      # Keras want input of type tf.float64 by default in depend on the processor
      state = tf.convert_to_tensor(state, dtype=tf.float64)
      next_state = tf.convert_to_tensor(next_state, dtype=tf.float64)
      actions = tf.convert_to_tensor(actions, dtype=tf.float64)
      reward = tf.convert_to_tensor(reward, dtype=tf.float64)
      done = tf.convert_to_tensor(np.float64(done), dtype=tf.float64)

      with tf.GradientTape() as tape:
        with tf.GradientTape() as tapebis:

          noise = np.clip(np.random.normal(0, self.__target_noise), -0.5, 0.5)

          next_actions = self.policy.get_target_action(state=(next_state + noise))

          next_actions = np.clip(next_actions, self.env.action_space.low, self.env.action_space.high)

          # compute Q(A', S')
          target_value = self.critic.computeTargetValue(state=next_state, actions=next_actions)

          target_value_bis = self.criticBis.computeTargetValue(state=next_state, actions=next_actions)

          target_value = tf.math.minimum(target_value, target_value_bis)

          # compute the target
          expected_value = reward + (1 - done) * self.__gamma * tf.cast(target_value, dtype=tf.float64)

          # compute Q(s, A) and QBis(s, s)
          q_value = self.critic.computeValue(state=state, actions=actions)

          q_value_bis = self.criticBis.computeValue(state=state, actions=actions)
          # now we are going to train the critics

          critic_loss = self.critic.computeLosses(value=q_value, target=expected_value)

          critic_loss_bis = self.criticBis.computeLosses(value=q_value_bis, target=expected_value)

          loss = critic_loss + critic_loss_bis

          self.critic_losses.append(loss)

        grad_bis = tapebis.gradient(loss, self.criticBis.valueNet.trainable_variables)

        self.criticBis.optimizer.apply_gradients(zip(grad_bis, self.criticBis.valueNet.trainable_variables))

      grad = tape.gradient(loss, self.critic.valueNet.trainable_variables)

      self.critic.optimizer.apply_gradients(zip(grad, self.critic.valueNet.trainable_variables))


      if step % 2 == 0:

          # train the actor
          with tf.GradientTape() as tape:
              # critic the action take by the actor at the state s
              new_actions = self.policy.evaluate_state(state=state)
              new_actions = tf.convert_to_tensor(new_actions, dtype=tf.float64)
              qValue = self.critic.computeValue(state=state, actions=new_actions)
              policy_loss = -tf.math.reduce_mean(qValue)
              self.policy_losses.append(policy_loss)

          # apply the computed gradient
          grads = tape.gradient(policy_loss, self.policy.policyNet.trainable_variables)

          self.policy.optimizer.apply_gradients(zip(grads, self.policy.policyNet.trainable_variables))

          self.copyNetworksToTarget()
```

Listing d

save : sauvegarde nos modèles.

```
def save(self):

    model_filename = "save/Policy_Model" + datetime.now().strftime("%Y%m%d-%H")
    target_model_filename = "save/Policy_Target_Model" + datetime.now().strftime("%Y%m%d-%H")
    self.policy.policyNet.save_weights(model_filename)
    self.policy.policyTargetNet.save_weights(target_model_filename)
    model_filename = "save/Critics_Model" + datetime.now().strftime("%Y%m%d-%H")
    target_model_filename = "save/Critics_Target_Model" + datetime.now().strftime("%Y%m%d-%H")
    self.critic.valueNet.save(model_filename)
    self.critic.targetValueNet.save(target_model_filename)
    model_filename_bis = "save/Critics_Model_Bis" + datetime.now().strftime("%Y%m%d-%H")
    target_model_filename_bis = "save/Critics_Target_Model_Bis" + datetime.now().strftime("%Y%m%d-%H")
    self.criticBis.valueNet.save(model_filename_bis)
    self.criticBis.targetValueNet.save(target_model_filename_bis)
```

Listing e

computeAndWritePolicyLoss : cette méthode nous permet de calculer la moyenne des erreurs sur un épisode et de les ajouter à notre graphique dans tensorboard.

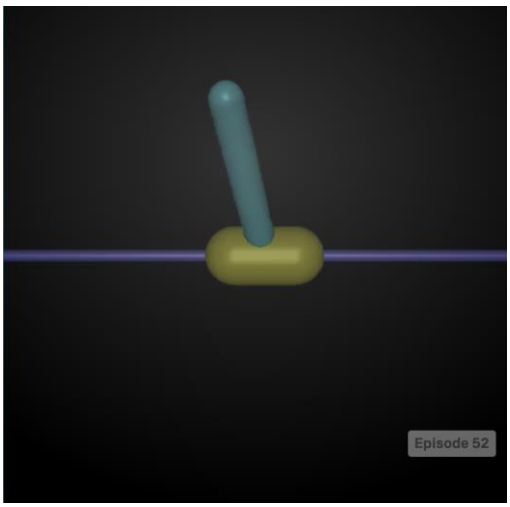
```
def computeAndWritePolicyLoss(self, episode):

    policy_loss_mean = np.mean(self.policy_losses)
    critic_loss_mean = np.mean(self.critic_losses)
    tf.summary.scalar('Policy loss', policy_loss_mean, step=episode)
    tf.summary.scalar('Critics loss', critic_loss_mean, step=episode)
    print("Episode {} policy loss : {} critic loss : {}".format(episode, policy_loss_mean, critic_loss_mean))
    self.critic_losses.clear()
    self.policy_losses.clear()
```

Listing f

Etant donné que notre robot possède de nombreux degrés de liberté, le temps d'entrainement et la puissance de calcul nécessaire à l'entrainement de notre agent sont trop importants, nous avons donc entrainé notre agent sur des environnements moins complexes.

Benchmark

RoboschoolInvertedPendulum-v1



Le problème du swing pendulaire inversé est un problème classique dans la littérature sur le contrôle. Dans cette version du problème, le pendule commence dans une position aléatoire et le but est de le faire pivoter pour qu'il reste droit.
Score Max : 1000

Figure r

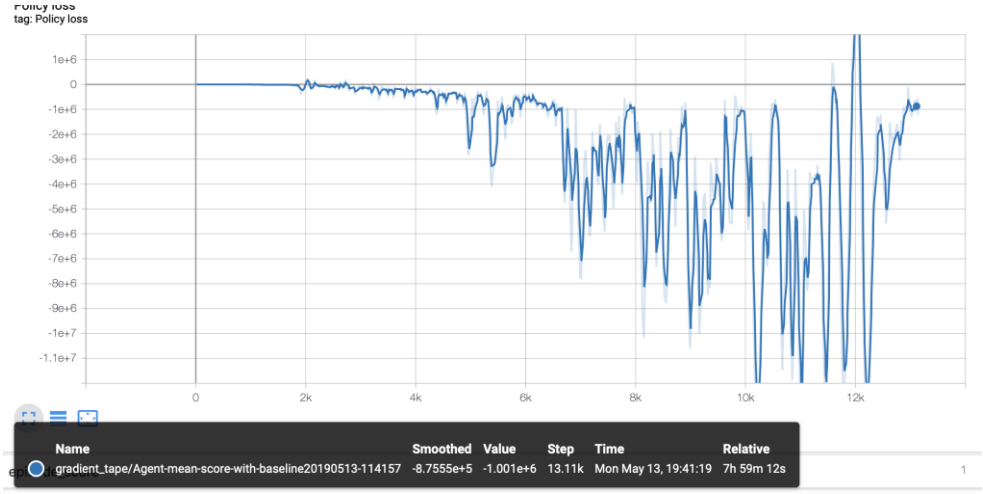


Figure s

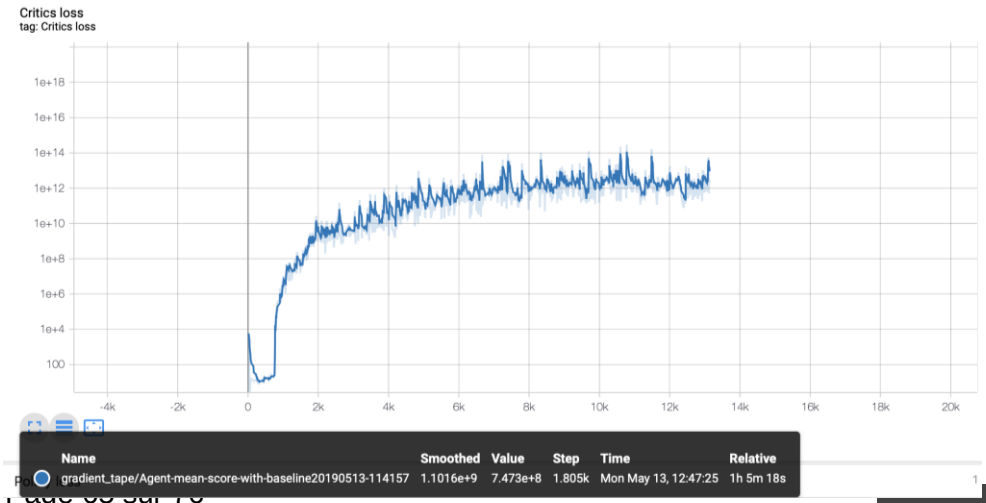


Figure t

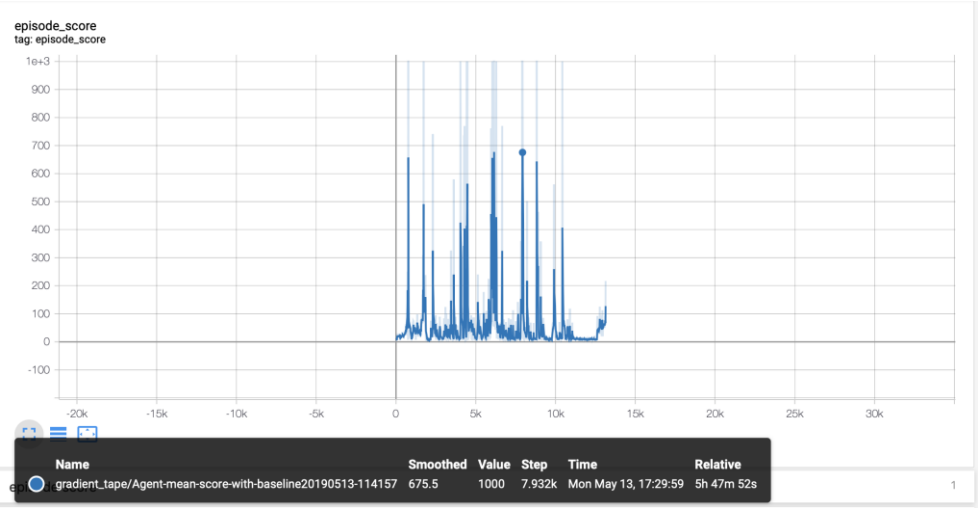


Figure u

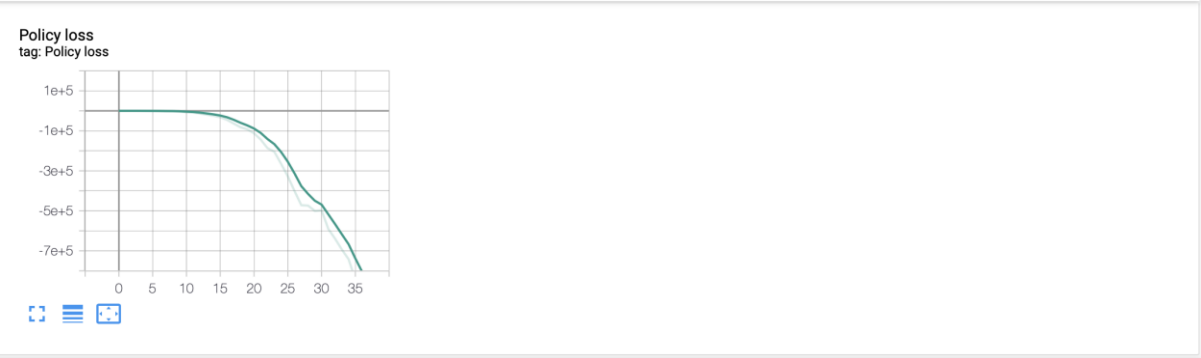


Figure v

MountainCarContinuous-v0

Une voiture est sur une piste unidimensionnelle, positionnée entre deux "montagnes". Le but est de monter la montagne à droite ; Cependant, le moteur de la voiture n'est pas assez puissant pour gravir la montagne en un seul passage. Par conséquent, la seule façon de réussir consiste à faire des allers-retours pour créer une dynamique. Ici, la récompense est plus grande si vous dépensez moins d'énergie pour atteindre votre objectif.

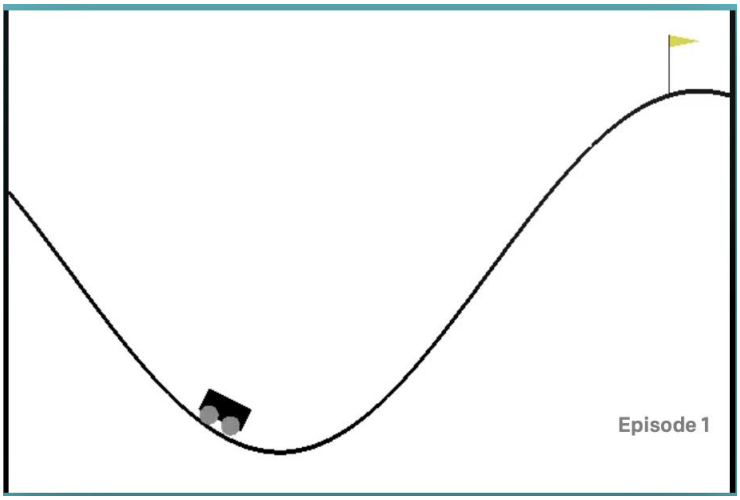


Figure w

Score Max : 90

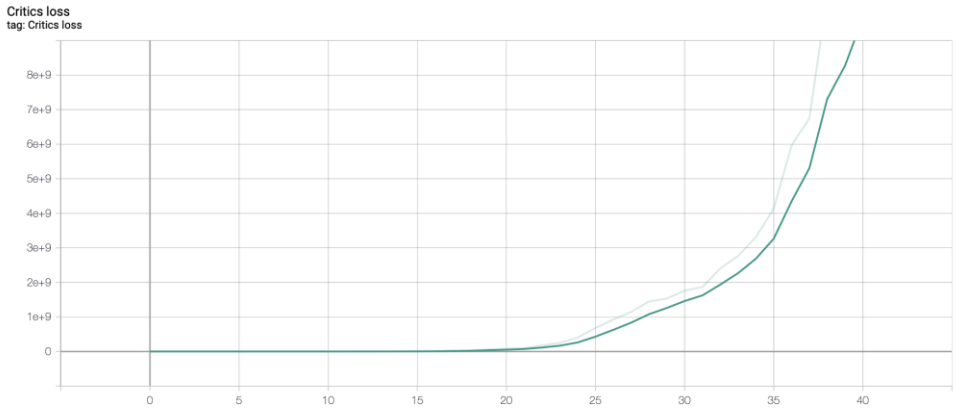


Figure x

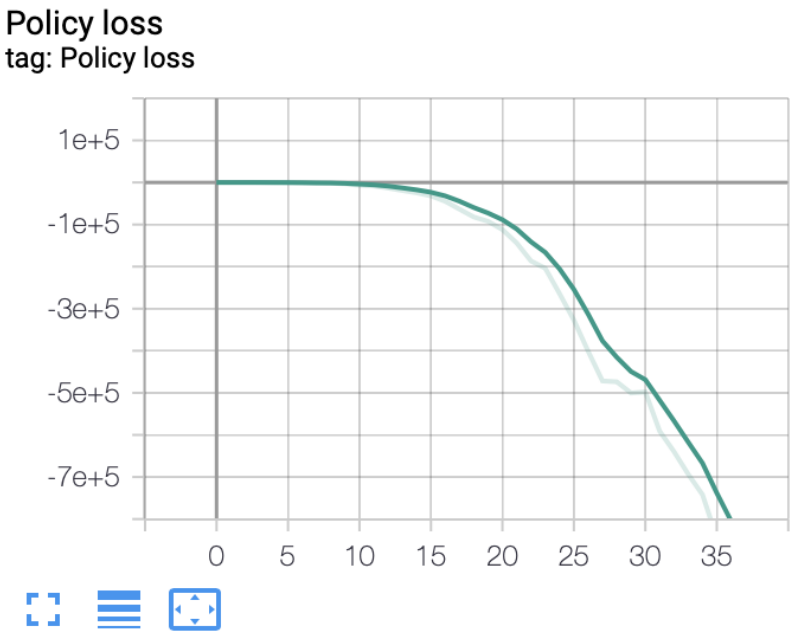


Figure y

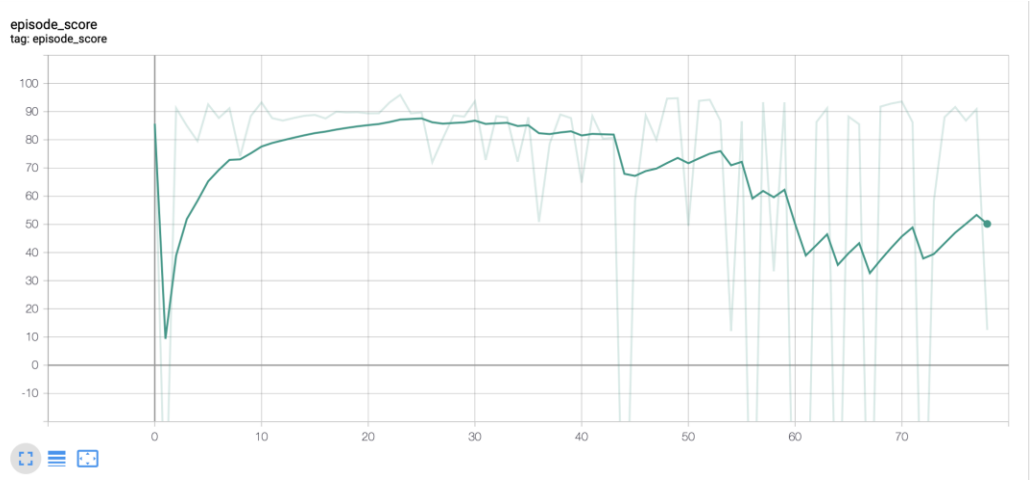


Figure z

RoboschoolInvertedDoublePendulum-v1

Version à contrôle continu à deux liaisons du problème classique du cartpole. Gardez le pendule à deux bras à la verticale en déplaçant le chariot 1-D. Semblable à la tâche MuJoCo InvertedDoublePendulum.
Score Max : 10000

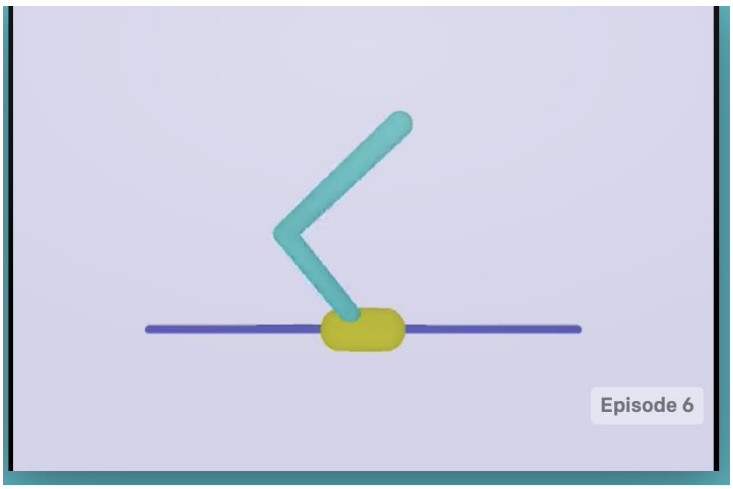


Figure aa

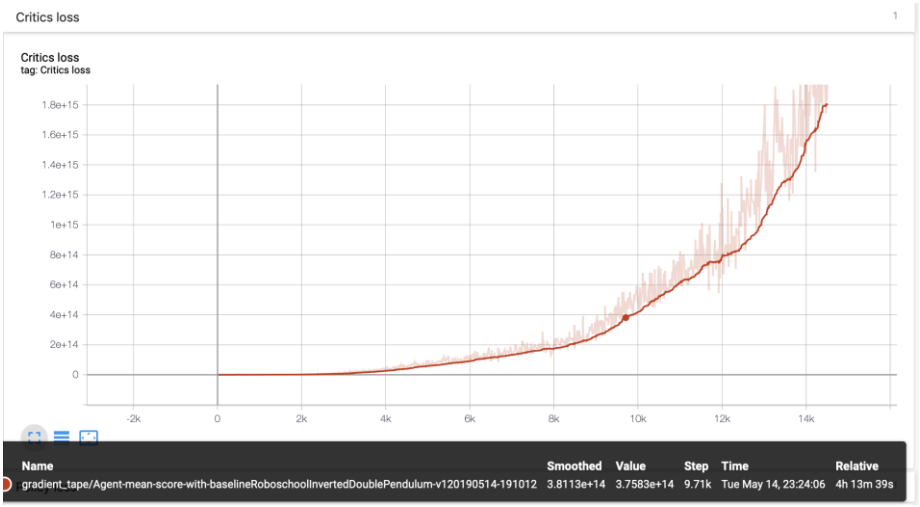


Figure bb

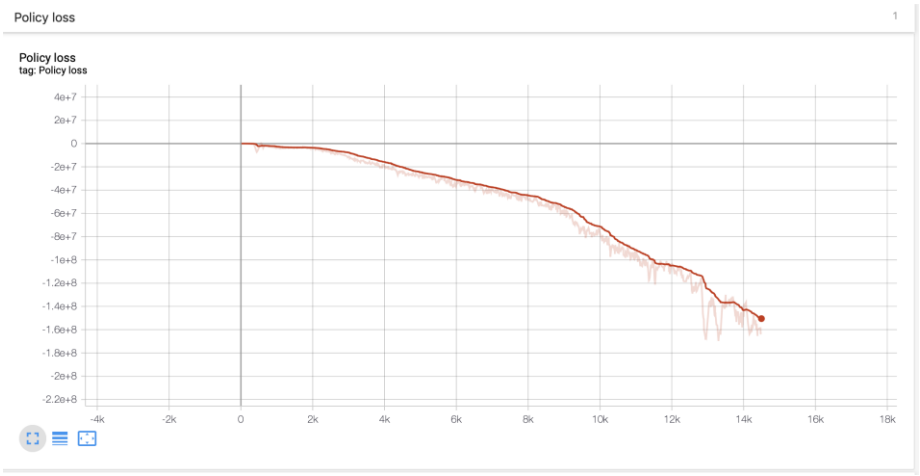


Figure cc

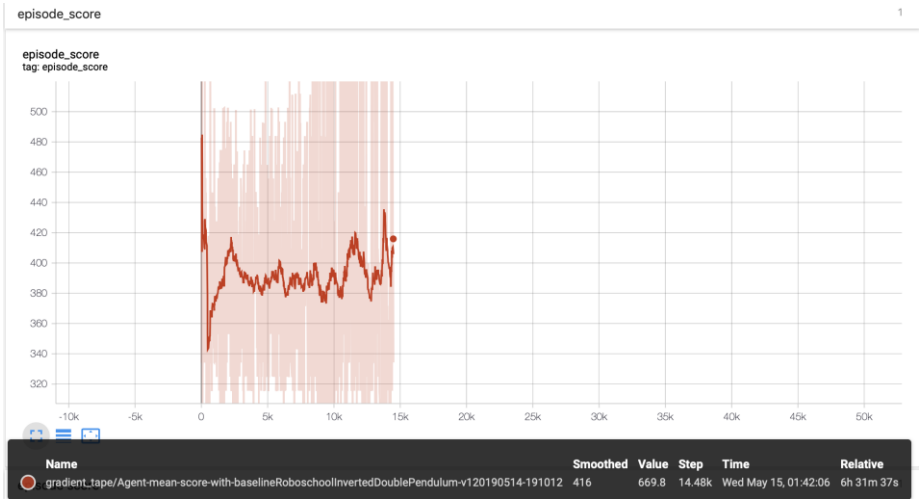


Figure dd

Durant la phase de test, l'IA a été entrainer durant 1 million d'itérations. Les hyperparamètres utilisées sont ceux qui ont donné les meilleurs résultats.

- régularisation : Non
- temps d'entrainement : 1 million d'itérations
- taux d'apprentissage : 1^-3
- coefficient de Polyak : 0.005
- activation: relu

Nous avons constaté que durant l'apprentissage, l'erreur de notre politique diminue, elle devient très négative. De l'ordre de -10^-6. Elle est égale à -Q(S, A). C'est la valeur que

notre critique attribut à notre action S. Cela signifie que notre critique trouve que les actions de notre acteur s'améliorent au fil du temps. Toutefois, l'erreur de notre critique augmentent à mesure que celui-ci apprend. Ce qui signifie que notre critique a de plus en plus de mal à atteindre la cible et devient moins précis dans son calcul de la qualité de l'action au fil de son apprentissage. Nous avons également remarqué que lorsque l'erreur de notre critique devient trop importante notre acteur se met à osciller. C'est le cas pour le **double inverted pendulum**. Pendant plus de 100k épisodes notre agent a oscillé. Il sera donc nécessaire que nous trouvions de meilleurs hyperparamètres afin de réduire la variance de nos critiques.

Résultats

Le projet s'est dans l'ensemble bien déroulé. La plupart des objectifs que nous nous sommes fixés ont été atteints. Seul le temps et la puissance de calcul nous ont manqué pour pouvoir, au moment du rendu de ce rapport, réaliser le pilotage du robot à l'aide d'une intelligence artificielle.
A présent, reprenons les objectifs que nous avons énumérés au début de ce rapport.
Les objectifs concernant l'acquisition de compétences en développement sur Unity et en machine learning son totalement remplis. En effet, nous avons eu l'occasion d'expérimenter de nombreux aspects dans ces deux domaines ce qui nous a fait énormément progresser.
Concernant la simulation, bien que certains comportements du simulateur soient parfois un peu éloignés de la réalité (notamment la gravité qui semble un peu faible et des collisions parfois un peu trop violentes) le résultat est proche de ce que l'on pourrait attendre d'un robot réel.
De plus, grâce au système de paramétrage et aux différents outils mis à disposition pour l'apprentissage d'une intelligence artificielle, la simulation est très polyvalente comme nous le souhaitions.
Le système de communication en utilisant basé sur l'envoi d'objets Json en utilisant le protocole UDP rend la communication très simple et rapide. Cela permet de pouvoir tester facilement nos scripts de pilotage du robot.
Enfin, concernant l'apprentissage automatique, notre travail de recherche nous a montré qu'il était possible de piloter un robot grâce à la méthode que nous souhaitions employer. De plus nous avons proposé des solutions diverses afin de résoudre des problèmes du même type mais simplifié (pour réduire le nombre de données et d'itérations nécessaires pour faire converger notre réseau). Nous n'avons malheureusement pas pu mettre en application notre modèle sur notre simulateur par manque de temps et de machines capables de faire tourner plusieurs dizaines d'heures le simulateur et le modèle d'apprentissage (nous avons estimé le temps de convergence du réseau neuronal à environ trois cents heures pour apprendre à marcher en simulateur).
Nous restons cependant convaincus que le client que nous avons développé fonctionnera lorsque nous aurons réussi à l'entrainer suffisamment car il a montré de très bons résultats sur des simulations simplifiées.

Conclusion

Aux vues des résultats de ces trois mois de travail, le projet qui nous a été donné à réaliser est une réussite. Les nombreux défis que nous avons surmontés nous ont énormément appris et nous sommes fiers de pouvoir vous rendre aujourd'hui le fruit de notre labeur ainsi que ce rapport qui, nous l'espérons, vous a intéressé et a été suffisamment clair pour que des personnes extérieures au projet puisse en comprendre les enjeux.
La réalisation de ce simulateur ainsi que les algorithmes d'apprentissage qui y sont associés nous a permis de mieux appréhender le monde de l'intelligence artificielle et du développement de logiciel. Cela a aussi renforcé notre volonté de continuer notre travail au sein de l'association PROROK sans qui ce projet n'aurait pas vu le jour. Nous tenons à remercier toutes les personnes qui ont rendu cela possible à commencer par Monsieur Dandoush, qui a accepté de nous laisser développer notre propre projet.

Conclusion de la présentation du simulateur

Le simulateur que nous avons conçu est donc complètement indépendant de l'intelligence artificielle qui lui est associée. Il propose différents paramétrages et différentes fonctionnalités afin de pouvoir facilement entrainer et tester différents modèles d'apprentissage. Bien qu'il pourrait bénéficier de nombreuses améliorations, il est complet et offre une simulation satisfaisante de la réalité. L'utilisation du moteur de jeu Unity permet de pousser encore plus loin sa polyvalence grâce à la possibilité de compiler le projet pour différents systèmes (Linux, Windows, MacOs, etc…). L'interface utilisateur a été pensée pour être la plus simple et intuitive possible afin de faciliter l'utilisation du simulateur par des personnes qui n'ont pas travaillé sur le projet et qui souhaiteraient développer leur propre modèle d'apprentissage. Grâce à la réalisation de ce projet, nous pourrons à présent développer un simulateur encore plus performant pour simuler le robot bipède que nous souhaitons réaliser, tout en évitant de répéter les différentes erreurs que nous avons commises lors du développement de celui-ci.

Conclusion de la présentation de l'intelligence artificielle

Notre modèle est capable d'apprendre une politique et de fournir en sortie des actions continues. Les améliorations qui ont été apportées à l'algorithme du TD3 ont permis de réduire l'erreur de nos critiques et ainsi d'améliorer notre politique. Le modèle est capable d'apprendre dans un temps raisonnable des tâches complexes. Toutefois, lorsque la dimension de nos observations augmente ou que le nombre de nos sorties augmente, ce temps d'apprentissage augmente considérablement, car la politique à apprendre est plus complexe. Il sera donc obligatoire d'acquérir des ordinateurs ayant plus de puissance ou d'entraîner notre agent dans un cloud afin que celui-ci apprenne plus rapidement.

Perspectives

Ce projet nous offre de nombreuses perspectives d'avenir en matière de pilotage automatique de robot et de simulation.
Bien que le projet soit officiellement fini nous allons continuer à le développer jusqu'à ce que nous ayons une première intelligence artificielle fonctionnelle.
Ensuite nous pourrons commencer le développement d'un simulateur pour robot bipède.
Grâce aux compétences acquises lors de ce semestre, nous pourrons concevoir une meilleure architecture de simulateur et améliorer grandement son réalisme.
Nous allons pouvoir aussi, grâce à cela, construire de meilleurs modèles de pilotage afin d'éventuellement, un jour, réaliser un réel robot piloté par intelligence artificielle.
Grâce aux compétences acquises par les autres membres de l'association, nous allons aussi pouvoir faciliter le travail de nos agents en développant des procédures automatiques de marche, détection d'obstacles et de gestion de l'équilibre.

Références

Addressing Function Approximation Error in Actor-Critic Methods Scott Fujimoto 1 Herke van Hoof 2 David Meger

Deep learning avec TensorFlow de Aurelion Géron