ENSTA BRETAGNE

Rapport de stage 2A
REMI GENOUX-LUBAIN

ROB26

27 octobre 2025

R. Genoux-Lubain

Introduction

Le but du stage a été de déployer un modèle de mouvement efficace et fiable basé sur l'apprentissage par renforcement pour le robot chien à roues de Linxai Tech. Pendant qu'une autre équipe se chargeait de développer un modèle de comportement pour le robot à quatre pattes (produit commercial de l'entreprise), il nous a été demandé de travailler sur un robot équipé de quatre roues au bout de ses pattes et de mettre en place les ajustements nécessaires pour qu'il puisse fonctionner. Nous devions obtenir des comportements adéquats pour un tel robot, c'est-à-dire délaisser la marche au profit de l'utilisation des roues dans les cas où c'était plus efficace, sans enlever la capacité de franchir des obstacles permise par les pattes.

Pour cela, nous avons cherché à atteindre deux objectifs complémentaires : d'une part, le robot devait bien se comporter dans son environnement virtuel en respectant les métriques et comportements attendus ; d'autre part, il fallait modéliser correctement les comportements réels du robot pour obtenir un entraînement réaliste en simulation.

Le code utilise pour cela une approche basée sur l'apprentissage par renforcement. Plus particulièrement, il repose sur un système Actor-Critic qui permet d'entraîner un robot virtuellement. Il dispose donc d'un système acteur, qui fournit les commandes, et d'un système d'évaluation, qui juge le comportement. Ce principe est rappelé et expliqué ci-dessous.

Nous verrons d'abord les mécanismes de base des réseaux de neurones et pourquoi ils sont efficaces, puis comment les faire s'améliorer pour adapter le robot chien à roues à son environnement virtuel. Enfin, nous présenterons la manière dont nous avons adapté les réseaux de neurones eux-mêmes pour améliorer le comportement virtuel et réel du robot chien. Dans cette dernière partie, l'objectif a été d'utiliser les différents modèles de chien à quatre pattes pour améliorer nos modèles de chien à quatre roues.

Table des matières

1 Présentation du stage 1
1.1 Choix du stage 1
1.2 Description de l'entreprise 1
1.2.1 Fournisseurs 2
1.3 Travail demandé 2

2 Description du robot 2
2.1 Cinématique 2
2.2 Capteurs 3
2.3 Architecture Logicielle 4

3 Fonctionnement du code 5
3.1 Rappel sur les réseaux de Neurones 5

4 Définitions et notions nécessaires 7
4.1 Réseau de neurones 7
4.2 Interprétation et influence des valeurs 8
4.3 Backpropagation et optimisateur 8
4.4 Actor-Critic 8
4.5 Vecteur de sortie 10
4.6 Récompenses 10

5 Travail demandé 11
5.1 Lancer des trainings 11

6 Modification de l'environnement (Première partie du stage) 12
6.1 Addition des quatres roues 12
6.2 Création de nouvelles récompenses 13
6.3 Phases de tests et d'améliorations 13
6.3.1 Tensorboard 13
6.3.2 Entraînement et détection de biais 14
6.4 Premiers résultats 15
6.5 Passage en terrain accidenté 15
6.6 observations et détermination d'un nouvel objectif 16

7 Modification des réseaux de neurones profonds (Deuxième partie du stage) 17
7.0.1 Concept de fusion 17
7.0.2 Réalisation et Résultats 18

8 Tests Réels (Sim2Real) 19
8.1 Robot mis à disposition et messages ROS 19
8.2 Résultats 19

9 Conclusion 20

ENSTA
IP PARIS

Liste des figures **21**

Bibliographie **22**

1 Présentation du stage

1.1 Choix du stage

Le stage nous a été proposé par Xiao Kai, fondateur d'une entreprise chinoise, qui avait réalisé sa thèse à l'ENSIETA. Son entreprise de robotique étant désormais basée en Chine, les enseignants de l'ENSTA nous ont transmis la proposition, à laquelle nous avons été trois à répondre positivement, avant de partir en Chine pour le réaliser. Le stage s'est déroulé de mai à août 2025 dans la ville de Shenzhen, au sud-est du pays.

La ville, qui compte presque 18 millions d'habitants, est connue comme un haut lieu de l'innovation électronique, robotique et logicielle, permise par le capitalisme contrôlé par l'État chinois. Ce système de fonctionnement n'existe qu'à certains endroits très spécifiques du pays : c'est donc au sein de ces pôles que se concentrent les entreprises de la tech chinoise. Shenzhen étant l'un de ces centres privilégiés, c'était un lieu idéal pour un stage en robotique. La ville évolue autour de ses entreprises, dans un climat compétitif et innovant. Nous avons donc intégré une entreprise à la pointe de la technologie en ce qui concerne son produit principal : le robot chien.

1.2 Description de l'entreprise

LINXAI TECH est une entreprise de robotique privée qui vise à développer et construire des robots intégrant des formes d'intelligence artificielle. Elle possède plusieurs centres de développement, l'un à Beijing, l'autre à Shenzhen. Elle dispose également de pôles de manufacture à grande échelle pour la production en série de ses robots.

Le pôle de développement de Shenzhen a pour principale activité la mise au point et la distribution des robots chiens, tandis que les autres activités sont gérées depuis Beijing.

L'entreprise a été fondée il y a six ans par plusieurs ingénieurs chinois, dont Xiao Kai, qui nous a proposé d'effectuer notre stage dans l'entreprise.

FIGURE 1 – Galaxy Twin towers, Linxai headquarters in Shenzhen, China

C'est une entreprise qui produit des robots de haute technologie à leur sortie, à des prix compétitifs par rapport au marché. Elle mise sur un développement en équipe réduite ainsi qu'une manufacture à bas coût et productive pour offrir un produit opérationnel et abordable.

1.2.1 Fournisseurs

Au cours de notre stage, nous avons eu la chance de pouvoir visiter les Au cours de notre stage, nous avons eu la chance de pouvoir visiter le centre de production d'un des capteurs du robot chien. La visite concernait l'usine de lentilles utilisées par le robot, la plupart étant destinées aux objectifs des différentes caméras.

Il a été marquant de voir à quel point la chaîne de production était automatisée et robotisée, ce qui la rendait d'autant plus intéressante pour des étudiants ingénieurs en robotique ! On observait des étapes réalisées par des robots à tous les niveaux de la production (et du bâtiment !). Qu'il s'agisse de bras robotisés manipulant des lentilles pour les trier, de chaînes complètes de transformation du verre ou de petits robots assurant le transport autonome des caisses, les robots étaient omniprésents et réalisaient un nombre impressionnant de tâches, des plus simples aux plus complexes.

Être plongé dans un environnement avec une telle concentration de technologie a été un véritable choc, révélant l'étendue de ce qu'il est aujourd'hui possible de réaliser — et suscitant une réflexion sur la place de l'humain au sein de telles organisations.

1.3 Travail demandé

LINXAI TECH a fait le choix de baser l'équilibre de son robot chien sur l'intelligence artificielle. Pendant qu'une équipe d'ingénieurs travaillait sur l'amélioration de ce système, il nous a été demandé d'adapter les algorithmes existants pour faire fonctionner une autre version du robot chien, comportant quatre roues à l'extrémité de ses pattes. Notre but était donc de faire apparaître les comportements qu'on pourrait attendre d'un tel robot : utiliser ses roues pour rouler sur le plat, les lever lorsqu'il faut tourner, et tirer parti des jambes pour franchir des obstacles.

L'objectif de notre stage était assez simple et pouvait être résumé en peu de mots, ce qui nous a laissé une grande liberté pour mettre en œuvre les solutions qui nous semblaient les plus adaptées.

2 Description du robot

2.1 Cinématique

Le robot chien commercialisé par LINXAI TECH s'inspire largement des architectures existantes sur le marché. Chaque patte est connectée au tronc par un ensemble de deux articulations principales, offrant trois degrés de liberté par patte, ce qui lui permet d'exécuter une gamme de mouvements variés, de la marche au saut.

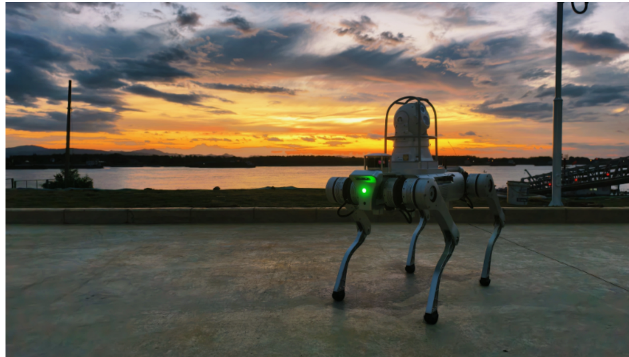

Figure 2 – Image du robot chien utilisé par LINXAI TECH

La première articulation, située à la hanche (hip joint), est un pivot dont l'axe de rotation est approximativement horizontal et colinéaire à l'axe d'avance du robot. Elle permet l'écartement latéral de la patte, ce qui est essentiel pour la stabilité et nécessaire pour les virages.

La seconde articulation, également située à la hanche, permet une rotation sur un axe horizontal perpendiculaire à la première. Elle simule le mouvement de la cuisse (thigh), produisant un mouvement de va-et-vient dans le plan vertical, qui est la base du mouvement de marche.

Enfin, la troisième articulation, située au milieu de la patte, régule la flexion et l'extension du segment inférieur, un mouvement comparable à celui du genou.

Chaque articulation est motorisée et commandée de manière indépendante. Dans le cadre d'un contrôle par IA, cette indépendance présente un double aspect. D'une part, le système n'a pas besoin d'apprendre les corrélations complexes entre les moteurs. Mais d'autre part, il devient plus difficile de restreindre les mouvements à ceux souhaités et d'éviter les comportements pouvant endommager le robot, comme les chutes ou les collisions entre les pattes.

2.2 Capteurs

Si tous les robots disposent de capteurs, ils ne sont pas tous équipés de la même manière. Certains capteurs, dits de base, sont communs à tous les modèles, tandis que d'autres, plus spécialisés (comme les lidars ou les caméras), améliorent leurs capacités. Enfin, certains capteurs (caméra infrarouge, détecteur de fumée...) sont ajoutés pour des missions spécifiques.

Parmi les capteurs omniprésents, on trouve la centrale inertielle, intégrée directement à la carte électronique du robot. Solidaire du tronc, elle mesure les vitesses de rotation et les accélérations de ce dernier. On peut aussi citer les codeurs incrémentaux (odomètres) présents sur chacun des douze moteurs de la configuration standard, qui renseignent sur l'angle de rotation de chaque articulation.

L'intégralité de notre stage s'est déroulée en utilisant exclusivement ces capteurs de proprioception. Nous n'avions accès qu'à ces données internes pour faire fonctionner notre robot.

2.3 Architecture Logicielle

Une équipe d'une douzaine de personnes, stagiaires inclus, a travaillé sur ce projet de robot chien boosté à l'intelligence artificielle. L'objectif était d'utiliser les données des capteurs pour apprendre au robot à marcher, franchir des obstacles, s'adapter à un terrain accidenté, tout en respectant une consigne de vitesse et de rotation.

Pour cela, le code s'appuie sur un principe bien connu en robotique : l'Apprentissage par Renforcement (Reinforcement Learning).

Pour simplifier, l'apprentissage par renforcement est une branche de l'intelligence artificielle qui se distingue par le fait qu'elle n'utilise pas de base de données annotée. Instead, elle apprend par essais successifs pour déterminer un modèle de comportement optimal.

Il est important de noter que l'entraînement des robots s'effectuait exclusivement en simulation. Les algorithmes n'apprenaient jamais à partir de données réelles, complexes à acquérir, mais à partir de milliers de données générées dans un environnement virtuel. L'avantage est de pouvoir exécuter des milliers de simulations en parallèle pour identifier le comportement optimal. Cependant, pour que ce comportement soit transférable au monde réel, la simulation doit être suffisamment réaliste.

Pour atteindre cet objectif, l'architecture logicielle se divise en deux parties principales :

— **L'environnement** : L'environnement désigne l'ensemble du processus de simulation virtuelle. Cette partie est indépendante de l'algorithme de RL. Elle doit être à la fois réaliste et économiquement en puissance de calcul pour permettre la parallélisation des simulations et multiplier les itérations d'apprentissage. L'environnement a une seconde utilité : en simulant les interactions du robot, il génère un jeu de données très riche pour chaque agent virtuel. On a ainsi accès à la position exacte d'une patte dans l'espace, son écart relatif, ou sa vitesse. Cette omniscience permet d'entraîner le modèle de manière robuste avec une surabondance d'informations, le préparant ainsi à fonctionner avec le minimum de données disponibles en conditions réelles.

FIGURE 3 – Environnement virtuel où évoluent les robots pendant leur entraînement

— **L'algorithme de RL** : L'algorithme de RL a pour rôle de générer et de faire évoluer un modèle de comportement en fonction des retours de la simulation. Le modèle lui-même est un réseau de neurones qui prend en entrée les données des capteurs et renvoie à chaque instant des commandes motrices. L'algorithme de RL utilise, quant à lui, toutes les informations à sa disposition pour évaluer les performances du modèle et le perfectionner.

Le fonctionnement plus poussé de cet algorithme est décrit dans la partie suivante.

3 Fonctionnement du code

3.1 Rappel sur les réseaux de Neurones

Abordons d'abord le principe dans sa globalité : un réseau de neurone est censé associer à un vecteur d'entrée un vecteur de sortie qui représenterait la réponse à une question donnée.
Par exemple, si un réseau de neurone est entraîné pour reconnaître les images comportant des avions, on aimerait qu'à une entrée d'image d'avion, il nous réponde si oui ou non l'image représente un avion. L'entrée ici est une image qui est représentée par un vecteur de donnée (en deux dimensions), et la sortie un vecteur qui est interprété par l'humain comme une réponse à la question (par exemple 1 pour un avion ou 0 pour autre chose).

Dans notre cas, c'est un peu plus compliqué, il faut adapter notre réseau de neurone à la question qu'on veut se poser, et les réponses à celles qu'on peut utiliser.

Explications : Nous avons à disposition un robot chien qui a des capteurs et des moteurs. Dans un monde idéal, nous aimerions qu'à chaque configuration de capteurs, **c'est à dire à chaque perception de la réalité du point de vue de notre robot,**

il puisse apprendre la bonne décision, et la prendre. Autrement dit, cela signifie que son réseau de neurone envoie les bonnes commandes à ses moteurs pour faire face à toutes les situation possibles et imaginable de manière adaptée.

Si l'on regarde bien, nous avons toutes les bases de ce qu'il faut comprendre pour ce rapport, toutes les questions, et déjà quelques réponses.

Maintenant, comment fait notre réseau de neurone pour « apprendre » la bonne décision ? Sans rentrer dans les détails pour l'instant, considérons un réseau de neurones et ses algotithmes comme une fonction qui peut se modifier elle même. Considérons aussi qu'on dispose d'un estimateur parfait qui arrive à dire exactement si un résultat est meilleur qu'un autre. Le principe de fonctionnement est alors simple : le réseau prend en paramètre le vecteur d'entrée, l'analyse, puis produit un vecteur de sortie qui induit un comportement sur notre robot (par exemple des commandes moteur). Ce comportement est ensuite évalué par notre estimateur. Après de nombreuses itérations, si un résultat est meilleur que les autres, l'estimateur fait en sorte que la fonction évolue pour se rapprocher des meilleurs résultats. C'est à dire, évolue "dans le bon sens".

Imaginons que le robot trébuche, on espère que le sens d'évolution soit vers une version du réseau de neurone qui induit un comportement où le robot lève plus les pattes.

Et ainsi, d'essais en essais, on convergerait vers une solution optimale, c'est à dire une solution qui correspondrait exactement au comportement souhaité, le meilleur comportement d'après l'estimateur.

La fonction optimale serait alors figée dans le marbre et à n'importe quelle situation, elle saurait **prendre** la bonne décision (= donner les bonnes commandes au moteurs) de manière à avoir le comportement souhaité.

On voit bien que derrière les mots que nous utilisons, il y a des questions plus que des réponses : Que veut dire « se modifier elle-même », comment cela se fait en réalité ? Comment construire un estimateur qui juge le comportement d'un robot ? Que veut dire « dans le bon sens » dans l'espace des paramètres ? Qu'est ce qu'une solution optimale ? Et en quoi est-elle optimale ? L'est-elle en réalité ?

Toute la construction du réseau de neurone doit prendre en compte ces problématiques et, dans le cas où la solution optimale est impossible à mettre en place, trouver les meilleures solutions pour réduire les erreurs et tout de même

4 Définitions et notions nécessaires

4.1 Réseau de neurones

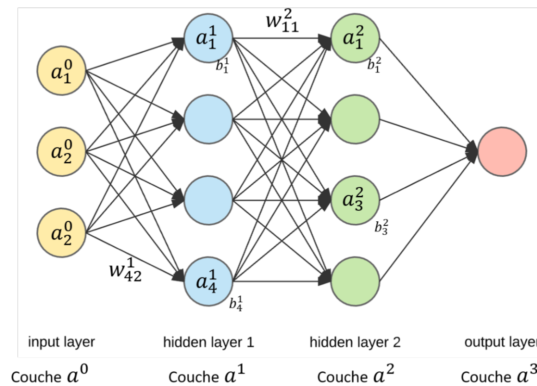

FIGURE 4 – Principe de fonctionnement d'un réseau de neurone en mode forward

Un réseau de neurones est une fonction qui prends en paramètre un vecteur d'entrée et qui produit un vecteur de sortie en passant par plusieurs pôles de calculs distincts. Le vecteur de sortie découle du vecteur d'entrée par une suite de calculs matriciels et de fonctions non-linéaires à la suite exécutées en chacun des pôles appelés neurones.

Plus précisément, un neurone est un vecteur de poids (= de valeurs). On multiplie ces poids avec un vecteur initial, puis on applique une fonction d'activation pour produire un résultat final. Les résultats finaux de chaque neurones peuvent ensuite être rassemblés pour former un nouveau vecteur ; nouveau vecteur qui peut être le vecteur initial d'autres neurones d'une couche différente pour produire un résulat final, c'est à dire un vecteur de sortie.

Appliqué au robot, les réseaux que nous utilisons sont des couches de neurones qui se succèdent pour calculer ensemble un vecteur de sortie. Le vecteur d'entrée de notre robot(342 valeurs qui représentent l'état de chaque moteur sur les 5 pas de temps précédent) est le vecteur initial de la première couche. Chaque neurone de cette couche aura donc 342 poids associés, un par valeur du vecteur initial. Au début de l'opération, chaque neurone exécute un simple calcul matriciel et donne une valeur en sortie (val1 x poids1 +...+ val342 x poids342). Le résultat passe par une fonction non linéaire appelées fonction d'activation. A noter que les poids sont différents pour chaque neurones, ce qui donne donc 512 résultats différents.

Les 512 résultats finaux des neurones sont rassemblées en un vecteur, qui formera le vecteur d'entrée de la deuxième couche. Ici, la deuxième couche sera donc formé de 256 neurones de 512 poids chacuns (puisqu'il y a les 512 résultats de la premiére étape à traiter). Et ainsi de suite sur les cinq couches de notre réseau de neurones (342, 512, 256, 128, 16) jusqu'à obtenir un vecteur de sortie de notre modèle qui contient 16 valeurs différentes.

Ce sont ces 16 valeurs différentes qui représente la sortie du modèle, c'est à dire les valeurs que nous allons utiliser pour commander notre robot virtuel et réel.

4.2 Interprétation et influence des valeurs

Si l'on comprends bien le processus présent dans les réseaux de neurones, on peut voir que chaque information de départ influe sur tous les neurones puisqu'un poids associé est présent sur chacun des neurones. Un poids fort associé à une valeur la rendra plus importante dans la sortie finale du neurones. Pour chaque neurone, certaine composante vont avoir plus d'influence que d'autres sur le résultat final du neurone. On espère en faisant cela, « mélanger » assez les informations et ce qui les lient les unes aux autres pour faire apparaître des liens de corrélations, de causes conséquences etc.. ...

Plus un réseau de neurones est étendu, plus on peut s'attendre à voir apparaître des nuances dans les réponses. ... Mais aussi de la complexité de traitement.

On peut donc intuiter qu'un réseau de neurones a la capacité de faire influer un paramètre de sortie précis en fonction d'un seul paramètre d'entrée donné ou de tous les paramètres d'entrée ou de seulement ceux qui sont pertinents.

Là où cela se complique, c'est lorsque qu'on demande au réseau de neurone d'avoir des comportements différents en fonction des situations. Puisqu'augmenter certains poids influencent l'ensemble de la sortie et pas uniquement une composante unique.

4.3 Backpropagation et optimisateur

Dans le code que nous avons utilisé, la modification du réseau de neurone en fonction du résultat obtenu (=backpropagation) se fait automatiquement en utilisant les données collectées.

Pour résumer son fonctionnement, il faut comprendre que notre réseau de neurones produit une série de commande moteur et induit un comportement sur notre robot simulé.

Un estimateur estime les erreurs associées aux mesures choisies (erreur de la vitesse par rapport à celle demandée, stabilité etc.. ...) et tout cela est rassemblé un score à notre robot. Score qui est directement utilisé par l'optimisateur pour modifier la structure du réseau de neurone qui a induit ce comportement.

Si un score est meilleur qu'un autre, l'optimisateur aura tendance à préférer le réseau de neurone qui a produit les « bonnes » commandes et modifier le réseau de neurone en conséquence.

Nous utilisons l'optimiseur d'Adam qui se charge de l'optimisation de notre réseau de neurone. Tout ce qui concerne l'évolution de notre réseau de neurones est regroupé dans la PPO (Proximal Policy Optimisation) associée au code que nous n'avons pas modifé. Nous avons seulement de l'influence sur les critères de notation de notre robot qui constituerons son score.

4.4 Actor-Critic

L'architecture Actor-Critic est la conception fondamentale du réseau de neurones qui pilote l'approche d'apprentissage. Il s'agit d'une implémentation spécifique du principe général décrit précédemment, où une seule fonction (le réseau de neurones) est entraînée sur la base d'une évaluation de ses performances.

Le principe est d'améliorer la fonction estimateur en utilisant un réseau de neurones complet pour estimer si une action prise par notre modèle d'action est correcte ou non.

Par conséquent, le modèle est divisé en deux parties distinctes :

1. **L'Actor** : C'est la partie du réseau qui décide quelle action entreprendre. Elle prend en entrée l'observation actuelle (le vecteur d'état) et produit en sortie le vecteur d'action (les 16 commandes des moteurs et roues). On peut le voir comme la « politique » du robot – sa réponse apprise à toute situation donnée. Son objectif est d'apprendre la politique optimale qui associe états et actions.
2. **Le Critic** : C'est la partie du réseau qui évalue les décisions de l'Actor. Elle prend en entrée le même vecteur d'état ainsi que l'action proposée par l'Actor, et produit une seule valeur : une estimation de la récompense totale future (le score) attendue en prenant cette action dans cet état. Son objectif est d'apprendre à prédire avec précision la valeur d'une paire état-action.

Le processus à chaque itération est le suivant : d'abord l'Actor observe l'état s et commande une action a. L'action est exécutée dans la simulation, conduisant à un nouvel état s' et générant une récompense immédiate r (calculée à partir de nos fonctions de récompense). Le Critic est alors consulté pour évaluer le résultat :

— Il estime la valeur de la paire état-action précédente : « Quelle a été la qualité de l'action a dans l'état s ? » (notée $V(s, a)$)
— Il estime aussi la valeur du nouvel état s' : « Quelle est la qualité d'être dans l'état s' maintenant ? » (notée $V(s')$)
— L'erreur de différence temporelle (TD-Error) est calculée. C'est le signal clé pour l'apprentissage :
$$\text{TD-Error} = \big(r + \gamma \times V(s')\big) - V(s, a)$$
où γ (gamma) est un facteur d'actualisation des récompenses futures.
— Une TD-Error positive signifie que l'action a a conduit à un meilleur résultat que ce que le Critic attendait. C'est une bonne surprise.
— Une TD-Error négative signifie que le résultat a été pire que prévu. C'est une déception.

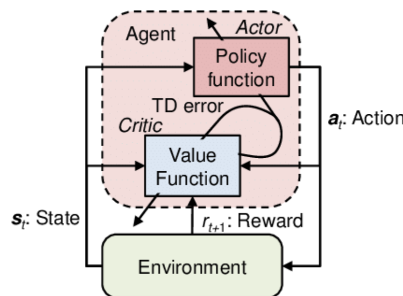

Figure 5 – Principe de fonctionnement d'un algorithme de RL en mode Actor-Critic

— Le Critic est mis à jour pour corriger son erreur de prédiction. Il apprend à fournir des estimations de valeur plus précises pour les paires état-action.
— L'Actor est mis à jour en utilisant la TD-Error comme signal directeur. Si l'erreur est positive, la probabilité de prendre l'action a dans l'état s est augmentée. Si l'erreur est négative, elle est diminuée. Les paramètres de l'Actor sont ajustés « dans la direction » suggérée par l'évaluation du Critic.

En essence, l'Actor est responsable de l'action, et le Critic est responsable du jugement de ces actions. Le retour continu du Critic permet à l'Actor de raffiner progressivement sa politique vers des actions qui produisent des récompenses prédites plus élevées, convergeant finalement vers le comportement optimal souhaité.

(Cette partie sur le système Actor-Critic a été générée par Intelligence Artificielle pour bien résumer la situation, je l'ai trouvée claire et précise et en accord avec des recherches ultérieures, j'ai donc décidé de la garder pour ce rapport)

4.5 Vecteur de sortie

Le vecteur de sortie est un vecteur de 16 composantes qui correspond aux actions que nous allons appliquer à chaque actionneur de notre robot. C'est donc une commande en couple pour chacun des actionneurs qui contrôlent les jambes (3 actionneurs par jambes) et une commande en vitesse pour les roues. Les actions sont récupérées du modèle de reinforcement learning par le programme qui gère l'environnement dans lequel évolue le robot et qui fait tourner la simulation.

Il est important de comprendre que ces valeurs ne sont pas directement reliées au robot pour le modèle d'intelligence d'artificielle. Il a seulement « observé » qu'en répondant de cette manière au vecteurs d'état qu'il recevait en entrée, le robot controllé de cette manière obtenait un bon score.

Ces valeurs sont donc interprétable par l'ingénieur et peuvent donner des pistes sur les améliorations à apporter au robot simulé ainsi qu'au modèle lui même pour obtenir de meilleures performance. Nous avons porté une attention particulière aux plages de valeurs ainsi qu'à leurs variations.

4.6 Récompenses

C'est la majorité du travail qui nous a été demandé et dont nous avons du nous acquitter. C'est tout simplement l'oeil que nous portons sur une version d'un modèle, la manière de faire comprendre à la machine ce que nous attendons d'elle. Les fonctions de récompense sont les critères qui vont faire partie de la notation de notre robot. Elles sont le résultat d'un simple calcul, généralement une erreur multiplié par un coefficient. La somme de ces récompenses sera ensuite donnée au programme de reinforcement learning pour prendre une partie importante de son évaluation. En somme, la quête des ingénieurs de reinforcement learning est assez simple : trouver les bons critères qui définissent un comportement optimal à leur yeux, et estimer l'importance qu'on doit leur donner pour obtenir un modèle performant, cohérent, et fiable. Par exemple, imaginons qu'on accorde une importance toute particulière à la vitesse horizontale de notre robot. Une bonne mesure serait l'erreur entre la vitesse demandée et la vitesse réelle sur cette axe. Et le coefficient associé doit être assez important par rapport aux autres pour qu'une erreur, même petite, pénalise le système de récompense et fasse comprendre à l'algorithme de manière numérique que c'est une erreur que les prochains modèles ont intérêt à faire diminuer pour obtenir un meilleur score. Une combinaison de ces critères forme un espace de contrainte, une espèce de carte de récompense où chaque combinaison de poids des neurones induit un comportement différent qui génère des récompenses différentes pendant la simulation et un score adapté.

Tout le but de l'algorithme est de trouver les paramètres qui optimisent la récompense. Tout le but de l'ingénieur est de trouver les récompenses qui débouchent sur les meilleurs comportements.

Il faut donc trouver les fonctions de récompenses qui caractérisent le mieux le comportement de notre robot :

Pour cela, il faut considérer les points sur lequel notre robot doit être évalué, dans quel mesure on doit les améliorer, et sous quelles formes cela peut exister. Bref, il faut prioriser.

Tout cela sachant qu'il y a plusiseurs manières d'évaluer un même critère. On peut étudier l'erreur en vitesse, mais aussi le carré de l'erreur, la dérivée de l'erreur ou encore l'intégrale de l'erreur, ou même faire une combinaison de tous ces critères etc.. .. Il existe de multiples possibilités d'évaluer un même critère, de multiples critères et encore plus de manières de les combiner. Car oui, même une fois que les choix des critères sont fait, il faut encore associer des coefficients à chaque critère pour estimer ce que le modèle prendra le plus en compte, et les affiner pour obtenir le modèle le plus performant.

Si l'on se rends compte que des comportements voulus restent innaccessibles malgré des changements de paramètres, il faut recommencer à construire des fonctions de récompenses différentes et tout recommencer.

```
1615
1616     def _reward_tracking_ang_integration(self):
1617         # Tracking of angular velocity commands (yaw) sur une fenêtre glissante
1618         ang_vel_error = torch.square(self.commands[:, 2] - self.base_ang_vel[:, 2])
1619         # Fenêtre glissante
1620         self.yaw_error_window[torch.arange(self.num_envs), self.yaw_error_ptr] = ang_vel_error
1621         self.yaw_error_ptr = (self.yaw_error_ptr + 1) % self.yaw_error_window.shape[1]
1622         sum_yaw = torch.sum(self.yaw_error_window, dim=1) * self.dt
1623         return torch.exp(-sum_yaw / self.cfg.rewards.tracking_sigma)
1624
1625     def _reward_lin_vel_z(self):
1626         # Penalize z axis base linear velocity
1627         return torch.square(self.base_lin_vel[:, 2])
1628
1629     def _reward_ang_vel_xy(self):
1630         # Penalize xy axes base angular velocity
1631         return torch.sum(torch.square(self.base_ang_vel[:, :2]), dim=1)
1632
```

Figure 6 – Exemples de fonction de récompenses qui mesures différentes erreurs

5 Travail demandé

5.1 Lancer des trainings

Chaque version de notre modèle courant est testé sur des centaines d'environnements en même temps. À la fin d'une certaine durée de test, tous les environnements qui fonctionnaient avec ce modèle calculent les récompenses associées à leur performances respectives, une moyenne est faite de l'ensemble des performances et c'est elle qui servira à modifier le modèle à travers l'algorithme de backpropagation. Ainsi, le modèle est jugée sur son action globale en considérant des centaines voir des milliers de situations, et pas seulement sur un environnement ou certains cas particulier pourraient fausser les résultats.

A la fin de chaque itération, le modèle est mis à jour et il est censé être plus performant. A puissance de calcul limité, il faut donc faire un juste arbitrage entre le nombre d'itération nécessaire pour juger de la pertinence d'un ensemble de coefficients/fonction de récompenses et le nombre d'environnement pour être certains que le modèle est

représentatif de ce qu'est « capable » l'ensemble coefficients/fonction de récompense.

On pourrait voir cela comme un déplacement dans notre espace de paramètres :

Augmenter le nombre d'environnement correspondrait à mieux choisir la direction à prendre à chaque fois tout en ralentissant la vitesse de convergence. Le baisser reviendrait à avoir une direction plus volatile, une convergence plus rapide, mais aussi un risque de non-convergence.

Si on a trop d'environnements, on ne sera pas assez rapide, chaque itération sera longue et il faudra énormément de temps pour atteindre l'objectif.

Si on en a pas assez, on risque qu'un robot qui se comporte mal fausse le calcul de récompense et dégrade la mise à jour de notre modèle.

Cela révèle la dualité qui est présente dans notre cas entre laisser un modèle tourner plus longtemps en espérant qu'il s'améliore. Et le faire tourner avec plus d'environnement en espérant qu'il prenne une autre direction. D'autant plus lorsqu'on veut seulement avoir une idée de la pertinence de nos nouvelles fonctions de récompenses, ou de nos nouveaux coefficients.

6 Modification de l'environnement (Première partie du stage)

6.1 Addition des quatres roues

Figure 7 – Photo du prototype à quatres roues de LINXAI TECH

Tout d'abord, pour que l'ensemble du code puisse fonctionner en considérant les changements, il faut qu'ils soient implémentés dans la simulation. Les roues que nous utilisions, au contraire des autres moteurs du robots, étaient commandées en vitesse de rotation et non en couple. Nous avons donc ajouté dans la simulation les roues ainsi que leur mode de contrôle dans le fichier .urdf, celui qui renseigne les géométries et les actionneurs du robot chien.

Toute l'architecture du code côté environnement a donc été modifiée pour ajouter quatres nouveaux acitionneurs, leur méthode de calcul de vitesse respectif, et assurer les

bonnes transmissions à l'algorithme de RL que nous n'avons pas eu à modifier dans un premier temps.

A ce stade, le robot fonctionnait en simulation, même si l'entraînement que nous lui fournissions n'était pas adapté à sa structure et à son mode de déplacement.

6.2 Création de nouvelles récompenses

Comme expliqué précedemment, toute la subtilité d'un programme de RL se concentre dans la création et la pertinence de la fonction d'erreur qui y est associée. En effet, c'est l'indicateur que l'algorithme doit minimiser, c'est, en somme, une valeur objective qu'on voudrait faire coller avec la valeur subjective qu'on a en tête (minimiser la fonction d'erreur = avoir un meilleur comportement). Tout le travail de l'ingénieur en informatique est de bien construire le couple fonction d'erreur/coefficient pour ajuster la fonction d'erreur au comportement souhaité.

Dans le cas de l'ajout de roues à notre robot, il a fallu trouver des récompenses adaptées à notre cas. Par exemple, nous avons voulu pénaliser la trop grande accélération de nos roues, et encourager leur utilisation dans les environnements plats. Mais nous avons aussi du encourager l'utilisation de la rotation par poussée différentielle lorsque la rotation est faible, et la pénaliser lorsqu'elle est forte, par préférence pour l'utilisation des jambes qui font tourner le robot plus efficacement. Une dizaine de fonctions de récompenses comme cela a du être crée ou modifiée.

Une fois que les récompenses sont crées, on leur adjoint leur coefficient. Pour des raisons de compréhension et d'efficacité dans le calcul, les récompenses sont généralement normalisées et les coefficients aux alentours de 1. En effet, donner un coefficient à une récompense indique le poids qu'elle aura dans l'évolution du modèle. Plus une valeur coefficient*recompense est importante pendant une simulation, plus l'algorithme aura intérêt à la faire baisser, ce qui a, encore une fois, pour conséquence d'éviter certains comportements et d'en favoriser d'autres.

6.3 Phases de tests et d'améliorations

6.3.1 Tensorboard

Tensorboard est une interface graphique pour suivre l'évolution des récompenses tout au long de l'entraînement. Cette interface permet d'essayer de comprendre ce qui a été important pour notre modèle lors de son entraînement ainsi que les points à améliorer.

Elle est surtout utile pour comprendre certains phénomènes de non convergence. Si une récompense baisse brusquement et que par la suite tous les paramètres commencent à se dégrader, alors on peut avoir une idée de quoi changer pour que ce cas ne se reproduise plus.

De même, si tous les paramètres n'évoluent plus ou plus beaucoup, alors le modèle a sûremnet convergé et n'évoluera plus de manière significative.

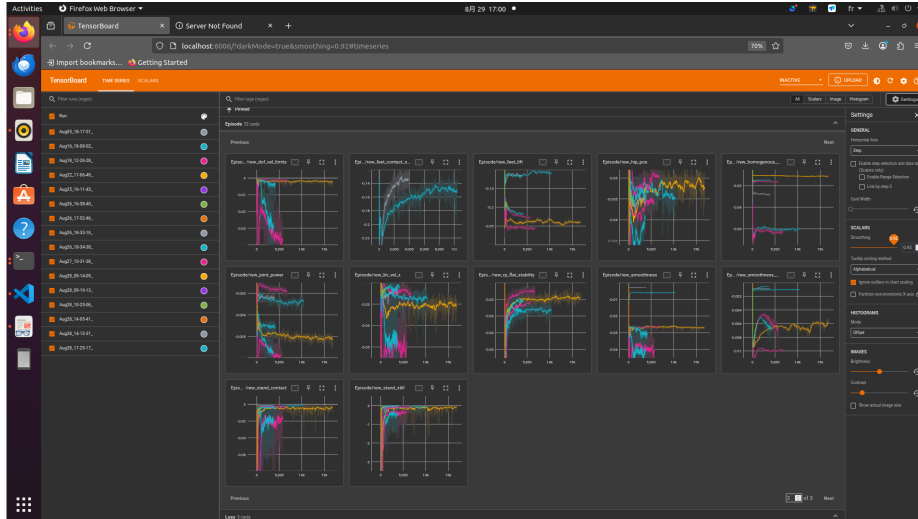

FIGURE 8 – Interface de visualisation de l'apprentissage sur Tensorboard

6.3.2 Entraînement et détection de biais

La première chose à faire avant de pouvoir juger un résultat pour notre robot, c'est de voir si le modèle a convergé vers un comportement qui minimise bien sa fonction d'erreur. On doit pouvoir observer une fonction d'erreur qui a été décroissante et qui s'est stabilisé autour d'une certaine valeur. Si cela n'est pas le cas, il faut soit admettre que la simulation n'est pas viable avec ces paramètres et que notre algorithme n'arrive pas à bien contraindre notre robot, soit qu'elle nécessite plus de temps pour finalement converger.

Sous réserve de convergence, il faut maintenant pouvoir faire tourner la simulation dans les mêmes conditions que l'entraînement, c'est à dire en utilisant le même environnement.

On nous a conseillé au sein de l'entreprise d'observer à l'oeil nu les déplacements du robot plutôt que de se focaliser sur les valeurs qu'elle peut nous renvoyer. Et il est vrai que la plupart des comportements que nous voulions éviter étaient facilement repérable.

Notre rôle lors de l'étape de contrôle du meilleur modèle est aussi de lui faire exécuter des mouvements qui n'arrivent pas ou rarement pendant l'entraînement, dans le but de détecter les biais auxquels le modèle a pu être exposé.

Il faut savoir que le modèle s'entraîne avec une fonction « curriculum » qui symbolise la difficulté auquelle est confrontée une génération de simulation. Si l'indice de curriculum est faible, cela signifie que les commandes envoyées sont peu extrêmes et/ou que le terrain est peu contrasté. Cela permet au modèle d'apprendre petit à petit plutôt que tout en même temps, ce qui le guide vers un minima global et non vers des parades qui pourraient lui faire baisser ses pénalités plus rapidement mais qui seraient des minimaux locaux.

Tester le robot avec des commandes qu'il n'a pas l'habitude de recevoir permet de vérifier s'il a bien généralisé les concepts qui sont censées être modélisée par les fonctions de récompenses.

Nous avons pu expérimenter des exemples de biais lorsque nous avons voulu tester la stabilité du robot lorsqu'il ne recevait aucune commande. C'est un cas qu'on rencontre

assez rarement pendant la simulation puisqu'on veut lui apprendre en priorité à se déplacer. Sans commandes envoyée, le robot commençait à tourner sur lui-même sans raison apparente. Il a fallu alors trouver des solutions, détecter les pénalités qu'il cherche à minimiser avec ce comportement pour trouver une autre manière de rendre celle-ci viable, ce que nous avons fait.

La conception et la modification des récompenses est une tâche très longue et fastidieuse puisqu'il faut plusieurs heures pour que les trainings puissent se réaliser, un temps où il est dur de faire des avancées sans avoir les résultats tangibles en face de soi.

6.4 Premiers résultats

Après avoir changé l'environnement dans lequel notre robot se déplaçait, nous avons pu lancer les premiers entraînements. L'objectif d'apprendre au robot à rouler sur terrain plat, et à se déplacer en utilisant principalement les roues.

Au bout de quelques semaines, le robot était capable en simulation de se déplacer comme nous le désirions. Il utilisait ses roues pour se propulser à la vitesse voulue lorsque la commande était sur son axe d'avance. et ses jambes pour translater sur son côté. De plus, on pouvait voir des comportements qu'on aurait pu s'attendre à voir émerger mais qui n'étaient pas directement codé par une erreur. Par exemple, il se stabilise en décalant son centre de gravité vers l'arrière lorsque le robot se met à avancer, et vers l'avant lorsqu'il se met à reculer.

Même si l'algorithme que nous utilisions était conçu pour favoriser la marche, nous avons réussi à effacer ce comportements pendant les phases d'avances ou de recul et à le conserver pour les déplacements latéraux. Les commandes dans toutes les directions du plans étaient respectée avec des erreurs très acceptables et nous n'avons pas réussi à le faire chuter même avec les commandes les plus violentes, ce qui démontre une capacité à se stabiliser dans n'importe quelles situations.

Toutes les commandes en rotation combinées ou non avec les commandes d'avance et de translation étaient de même respectées lui permettant de réaliser des mouvements complexes de manière controlée.

Enfin, en l'absence de commandes, le robot se stabilisait et ne réalisait aucun mouvement superflu. Nous avons réussi à obtenir un comportement satisfaisant sur tous les points en simulation.

Cependant, les robots chiens à roues ne sont pas fait pour se déplacer sur des terrains plats, là où un véhicule classique avec des roues à un avantage certains. Ils sont plutôt fait pour se déplacer en terrain accidenté, ou pour franchir des obstacles tels que des escaliers ou bien des terrains rocailleux. C'est pourquoi ceux qui comportent des pattes classiques sont d'abord entraîné sur le plat avant d'être rapidement mis à l'épreuve sur des terrains plus difficiles.

6.5 Passage en terrain accidenté

Pour ce faire, encore une fois, on ne laisse pas notre robot se débrouiller tout seul dans un terrain trop dur pour lui. En effet, nous avons utilisé la même fonction de curriculum qui servait à agrandir l'amplitude des commandes pour le terrain plat pour augmenter la difficulté du terrain rencontré.

Basiquement, on a un type de terrain (escaliers, terrains accidenté etc. . .) et on augmente sa verticalité pour le rendre plus difficile à franchir au fur et à mesure que notre algorithme apprends.

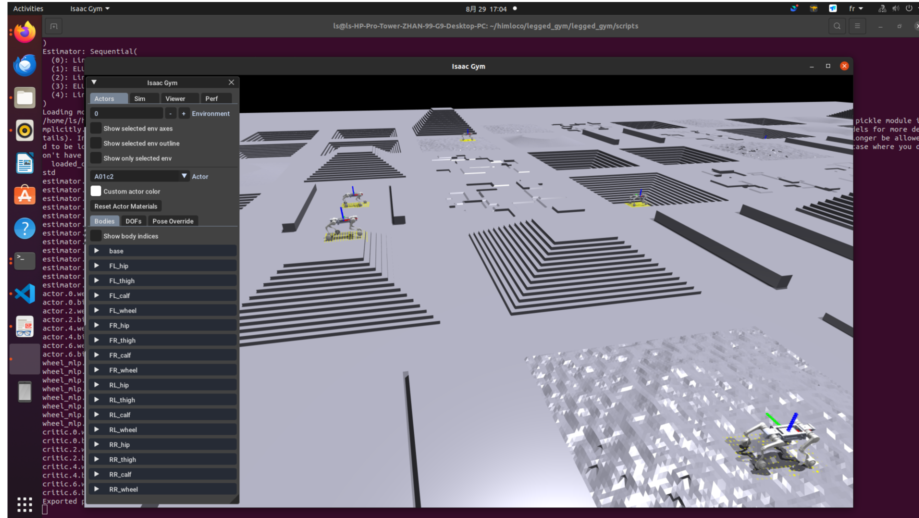

FIGURE 9 – Terrain virtuel accidenté avec plusieurs difficultés

Pour le passage en terrain accidenté, il a fallu concevoir de nouvelles fonctions de récompenses et manipuler de nouveaux leurs coefficients associés. Ce qu'on demande à notre robot étant assez différent en terme de physionomie du mouvement, il était clair que ces changements semblaient inévitables. Toute la complexité de la tâche était de ne pas faire oublier au robot la bonne utilisation de ses roues. Une solution était de bloquer les roues pendant les phases de déplacements difficiles et à changer de modèle pendant les phases de déplacements sur le plat. Ce n'est pas la solution que nous avons privililégiés. Nous voulions un unique modèle qui puisse fonctionner dans tous les types de terrains et adapter son comportement en fonction des commandes reçues et du terrain perçu.

Cela demande donc de pouvoir faire un usage limité des jambes sur certains terrains et un usage limité des roues sur d'autres. Une telle disparité des comportements désirés rends l'apprentissage plus difficile et plus instable.

6.6 observations et détermination d'un nouvel objectif

Il a été beaucoup plus dur d'avoir des résultats moyens en terrain accidenté que de bons résultats en terrain plat. Au fur et à mesure des entraînements, nous avons constaté la difficultés qu'a notre robot pour bien comprendre l'environnement qui l'entoure et pour réagir en conséquence. En effet, en ce qui concerne le terrain plat, il est invariant dans toutes les directions, pour ce qui est des escaliers par contres, qui sont présent sous formes de pyramides telles que celles des incas, de plus en plus escarpées. Il est dur de prévoir les reliefs qui sont présents dans toutes les directions.

On passe donc d'un modèle qui est capable de prévoir ce qui va lui arriver en conséquence d'un mouvement ou d'un autre, à un autre qui doit être autrement plus

flexible. La réaction du robot après un mouvement étant à chaque fois différente, il doit pouvoir s'adapter à toutes les situations différentes pour réagir au mieux, ce qui le rends d'autant plus complexe à entraîner.

On a donc convergé dans nos recherches de fonctions/coefficients vers une version du robot qui continuait à rouler mais qui n'arrivait pas à utiliser ses pattes correctement pour franchir les obstacles qui dépassaient la hauteur de sa roue.

On aurait pu lui mettre des roues de tracteur pour résoudre ce problème, mais nous avons optés pour une solution plus élégante. Nous avons pris la décision de changer le fonctionnemnet complet de notre réseau de neurone pour le modifier plus profondément. Nous avons entrepris de séparer les réseaux de neurones qui controllaient les roues et

ceux qui contrôllaient les moteurs des pattes dans le but de pouvoir exploiter les modèles fonctionnels des robots classiques.

L'idée est d'initialiser les paramètres du modèle des pattes dans la configuration d'un modèle fonctionnel et qu'on rajoute un autre modèle qui s'occupe des roues par dessus. On espère qu'avec un peu d'entraînement, les deux modèles peuvent synergiser tout en conservant la capacité de marcher du modèle classique.

Nous avons donc fixé un nouvel objectif, permettre à notre modèle de s'initialiser en partie avec un modèle déjà existant pour controller les pattes et espérer que cette technique puisse mener à des améliorations. Cet objectif avait un double avantage, permettre à notre robot d'égaler les performances permises par le robot classique et profiter des amélioraitons du robot classique pour directement les implémenter (en utilisant ce modèle comme modèle initial) dans le robot à roues.

7 Modification des réseaux de neurones profonds (Deuxième partie du stage)

7.0.1 Concept de fusion

Pour comprendre le concept de fusion, il faut savoir qu'un modèle est entraîné pour la tâche qu'on lui a demandé de réaliser et pas autre chose. On ne peut évidemment pas initialiser notre réseau de neurone entier comme le précédent, et attendre de lui qu'il comprenne que des valeurs supplémentaires sont rajoutées, et qu'elles doivent être manipulées différemment sans trop modifier ce qui existe déjà.

Pour réaliser l'initialisation d'une partie de nos paramètres par des modèles déjà existant, il faut donc séparer le réseau de neurone Actor destiné au calcul des commandes moteurs en deux réseaux de neurones distincts. L'un sera initialisé de manière standard et renverra les commandes associées au roues et l'autre sera intialisé avec les données d'un modèle déjà existant et renverra les commandes moteurs pour les pattes.

Ensuite, on doit encore modifier les informations que chaque modèle prends en entrée. Encore une fois, on doit séparer les données qu'on donne en entrée aux différents modèles pour s'adapter aux données que le modèle à quatre pattes avait à disposition. Nous sommes donc contraint d'isoler le fonctionnement des pattes du fonctionnement des roues : le modèle qui fait des choix pour les pattes n'a pas connaissance des actions précédentes des roues.

Concernant le modèle spécifique des roues par contre, on a pas de telle limite, on peut donc lui donner en entrées les anciennes actions des pattes en plus des données capteurs disponibles pour les deux modèles.

En faisant cela, on pourrait penser qu'on perds une partie de la connectivité qu'il y a entre les roues et les pattes, condamnant les roues à s'adapter aux pattes et non l'inverse. Cependant il existe encore un lien entre les deux modèles : l'impact qu'ils ont en simulation.

Nous comptions sur un mécanisme présenté plus haut pour assurer le transfert d'information par voie indirecte : la backpropagation. En effet, le but n'est pas d'initialiser les modèles et de ne plus les modifier par la suite. Le but est de les faire s'entraîner de nombreuses fois en espérant qu'ils n'oublient pas tout de la manière dont ils ont été initialisés, d'autant plus qu'on sait cette manière plutôt efficace.

En s'entraînant, les deux modèles ont un impact conjoint sur la simulation. Un bon comportement sera toujours estimé de la même manière par l'estimateur qui lui n'a pas été modifié et inversement. Donc on peut espérer que le réseau de neurone qui s'occupe des roues soit modifié, par les algorithmes de backpropagation en fonction du comportement du robot entier.

Si notre estimateur est efficace, on peut donc espérer voir apparaître un comportement qui favorise des réactions communes des pattes et des roues pour réduire la fonction d'erreur. En soi, on cherche à avoir deux comportements complémentaires tout en s'ignorant, ce qui est loin d'être impossible.

Les deux vecteurs de sorties de chacun des modèles sont donc concaténés et réindexé pour atteindre le même objectif qu'un modèle ayant été complètement construit à partir d'un seul modèle d'actor.

7.0.2 Réalisation et Résultats

La plus grande difficulté que nous avons rencontrée est celle de coller aux différents formats que les différents algorithmes utilisaient pour les redistribuer correctement aux bons réseaux de neurones. Le même défi était à relever pour redistribuer les valeurs en sortie des modèles, dans le bon ordre, et sans intérférer dans les autres fonctions disponibles de l'algorithme.

Une fois ces difficultées levées, les résultats ont été meilleurs très rapidement.

Là où auparavant notre robot n'arrivait pas à franchir la majorité des escaliers, sa nouvelle version n'échouait qu'au niveau le plus difficile.

Du point de vue du comportement, le robot avait effectivement bien gardé l'avantage de départ qu'on lui avait donné pour la marche et le franchissement d'escaliers. De plus, son entraînement avait ajouté à ce comportement la capacité à rouler quand le terrain le permettait.

La nouvelle architecture a donc été bénéfique du point de vue des performances du robot, elle lui a permis de mieux se comporter et d'intégrer plus facilement les roues dans son architecture. De plus, nous avons réussi à créer un système qui puisse profiter des avancées futures sur le robot classique pour pouvoir s'améliorer rapidement.

On peut tout de même noter que les performances en terrain accidenté sont moins impressionnantes que celles obtenues par un robot chien à patte qui lui franchit avec

certitude et facilité les obstacles les plus difficiles. Il aurait fallu plus de tests et d'entraînement pour trouver le bon axe et tirer profit de toutes les capacités du modèle fusionné.

8 Tests Réels (Sim2Real)

8.1 Robot mis à disposition et messages ROS

Nous avions à notre disposition un robot qui était déjà équipé de roues et qu'on nous a laissé libre d'utiliser. Le robot est un robot classique de LINXAI TECH comportant quatre patte et une roue au bout de chaque pattes. Les roues sont commandées avec un nœud ROS qui avait été créé avant notre arrivée. Ce nœuds actionne les roues en conséquence d'un message contenant les vitesses de rotation en tour par minute, message qui doit être indépendant des messages qui contrôlent l'ensemble des autres articulations. Nous avons donc modifiés la structure du nœud ROS pour pouvoir envoyer et recevoir les différents messages une fois que les actions étaient extraites du modèle qui faisait tourner le robot. Ces messages étaient légèrement transformés pour correspondre aux différentes unitées utilisées par le robot et envoyées aux actionneurs par la suite.

A noter que le robot pouvait fonctionner sous deux mode différent, un mode RL où toutes les instructions provenaient du modèle d'intelligence artificielle intégrée au robot. Et un mode Classique, qui fonctionne avec une équation d'état et qui permet au robot de marcher dans sa forme initiale.

8.2 Résultats

Sur les quelques fois où nous avons pu tester notre modèle sur le robot réel, nous avons pu observer des comportements assez différents de ceux simulés. Certains actionneurs avaient l'air de réagir de manière beaucoup trop forte par rapport aux actions qui étaient envoyées. Nous sommes alors tombé face à un problème plus complexe qu'il n'y paraît. En effet, sur un problème classique, une simple fonction de gain suffirait à au moins essayer de comprendre de combien le décalage est présent, et plus particulièrement sur chaque actionneurs.

Plus spécifiquement, il nous semblait que les roues avaient des valeurs trop élevées ainsi que les articulations des hanches. Tandis que celles des coudes peinaient à faire se lever les roues dont le poids était peut être un peu lourd.

Nous ne pouvions donc pas utiliser une simple fonction de gain puisque les valeurs d'entrées du modèle sont les positions des moteurs et que la fonction qui génère les actions, le réseau de neurone est tout sauf linéaire. Modifier les valeurs d'entrée et de sorties n'a engendré aucune amélioration notable, provoquant au contraire des comportements soudains, inattendus et désordonnés.

Il faut comprendre que le modèle a été entraîné sur la base d'une physique précise et a développé des comportements qui s'adapte à elle. Changer sa perceptiondu réel, c'est lui faire croire à une physique différente, une qu'il n'a pas été entraîné à concevoir. Il est donc très difficile de faire marcher un modèle s'il ne marche pas de prime abord. La seule solution était de réaliser de nombreux aller retours entre la simulation et le réel pour améliorer la simulation et la faire coller plus à ce que l'on pouvait observer. Nous

n'avons malheureusement pas eu le temps de développer plus cette partie de notre stage. En effet, un léger accident a mis en défaut un déréglement d'un des moteurs, qui se décalibrait de manière récurrente, le rendant inopérable, même en mode Classique. Cela necessitait un changement de moteur ou une réparation plus profonde, qui n'a pas eu le temps d'être exécutée avant la fin de notre stage. Nous n'avons donc pas eu l'occasion de complètement franchir le gap SimToReal même si quelques améliorations nous ont permis de voir notre robot rouler, se déplacer et marcher un petit peu avec notre modèle qui fonctionnait sur le plat.

FIGURE 10 – Casse moteur de notre robot pendant sa phase de mise au point

Durant la dernière semaine de notre stage nous avons essayé de mettre en place une confirmation de notre algorithme en faisant de nouveaux tests sur une autre simulation. Mais l'apprentissage de Gazebo fut un peu trop long pour avoir de vrais résultats.

9 Conclusion

Table des figures

1 Galaxy Twin towers, Linxai headquarters in Shenzhen, China 1
2 Image du robot chien utilisé par LINXAI TECH 3
3 Environnement virtuel où évoluent les robots pendant leur entraînement . 5
4 Principe de fonctionnement d'un réseau de neurone en mode forward . . . 7
5 Principe de fonctionnement d'un algorithme de RL en mode Actor-Critic 9
6 Exemples de fonction de récompenses qui mesures différentes erreurs . . . 11
7 Photo du prototype à quatres roues de LINXAI TECH 12
8 Interface de visualisation de l'apprentissage sur Tensorboard 14
9 Terrain virtuel accidenté avec plusieurs difficultés 16
10 Casse moteur de notre robot pendant sa phase de mise au point 20

Références