

Mémoire de fin d'études

Pour l'obtention du diplôme d'ingénieur

Spécialité : Systèmes Embarqués (SE)

Marche robotique quadrupède par le machine learning

Réalisé par :

M. OLIVARES David

Encadré par :

M. CHENCHANA Bilel (Capgemini)

M. CHESNAIS Olivier (ECE)

Soutenu le XX Septembre 2021, Devant le jury composé de :

Mme. : - Présidente

Mme. : - Examinateur

M. : - Rapporteur

Année universitaire : 2020/2021

Table des matières

Liste des sigles et acronymes	III
Table des figures	IV
1 Contexte et présentation générale	1
1.1 Altran puis Capgemini Engineering	2
1.2 Le département recherche	2
1.3 Mon projet et son équipe	2
1.3.1 Introduction	2
1.3.2 L'équipe	3
1.3.3 Tâches du robot	3
1.3.4 Mobilité du robot	5
1.3.5 La vision du robot	7
1.3.6 Diagramme de Gantt	8
2 Présentation de la politique de responsabilité sociale	9
3 État de l'art	10
3.1 Contexte et problématique	11
3.2 Le Reinforcement Learning	11
3.2.1 Généralités et Définitions	11
3.3 Principaux algorithmes existants	14
3.3.1 Policy Gradient (PG) algorithms	14
3.4 Améliorations du Policy Gradient	15
3.4.1 Advantage Actor-Critic	15
3.4.2 Proximal Policy Optimization (PPO)	16
3.5 Apprentissage et récompenses	17
3.6 Veille scientifique	18
4 Travaux et réalisations	24
4.1 RéPLICATION des résultats et limites du papier choisi	25
4.2 Environnement et outils	26
4.3 Réalisations	28
4.3.1 Adaptation au Transfer Learning	28
4.3.2 Rendre le réseau pilotable	28
4.3.3 La dérive du robot	32
4.3.4 Manoeuvrabilité du robot	35
4.3.5 Perspectives et travaux futurs	36

Table des matières

Annexes	41
----------------	-----------

Liste des sigles et acronymes

IA	<i>Intelligence Artificielle</i>
ML	<i>Machine Learning</i>
DL	<i>Deep Learning</i>
RL	<i>Reinforcement Learning (Apprentissage par Renforcement)</i>
MDP	<i>Markov Decision Process (Processus de Décision Markovien)</i>
PG	<i>Policy Gradient (Gradient de Politique)</i>
PPO	<i>Proximal Policy Optimization (Optimisation de la Politique Proximale)</i>
ESN	<i>Entreprise de Services du Numérique</i>
SUR	<i>Smart Universal Robot</i>
R&D	<i>Recherche & Développement</i>
CV	<i>Computer Vision</i>
HUFII	<i>Hybrid Unit For Industrial Inspection</i>
SB2	<i>Stable Baselines 2, librairie python pour le Reinforcement Learning</i>

Table des figures

1.1	Robot HUFII et ses modules interchangeables (de gauche à droite) : bras multi-fonctions, brase télescopique, drone (visualisation Blender)	4
1.2	Robot HUFII en contexte d'inspection industrielle (visualisation Blender)	4
1.3	Robot HUFII, inspection peinture sur une aile d'avion (visualisation Blender)	4
1.4	Exemple d'inspection, dommages de surfaces	5
1.5	Robot HUFII en mode quadrupède	5
1.6	Robot HUFII en mode locomotion à roues	6
1.7	Roues mécanum dans l'épaule du robot HUFII	6
1.8	Exemples d'estimations de poses	7
1.9	Exemple d'identification de défauts métalliques	7
1.10	Diagramme de Gantt représentant la planification de mes tâches	8
3.1	Schéma explicatif d'un Markov Decision Process	12
3.2	Equation de Bellman (Value Fonction) associant une valeur à chaque état du MDP	13
3.3	Equation de Bellman (Q-value) associant pour chaque couple (s,a) une valeur "Q" représentant la valeur d'une action à prendre dans un état donné.	13
3.4	Score des essais noté J. Correspond à la moyenne des récompenses obtenues au cours des essais en suivant la policy π_θ	14
3.5	Gradient de la fonction score (objective function) de la policy π	14
3.6	Description de l'algorithme de gradient ascendant pour optimiser la policy	15
3.7	Advantage Function soutrayant la Q-value du couple (état, action) avec la valeur moyenne de l'état	15
3.8	Réécriture de l'Advantage Function	16
3.9	Réseau absorbant le reality gap (Actuator Net)	19
3.10	Schéma architectural Master - Student	20
3.11	Architecture globale : planner et controller	21
3.12	Démonstration en simulation (pas : ronds sur le sol, lignes : trajectoires) . .	21
3.13	Sous-récompenses de plusieurs parties du robot	22
3.13	Sous-récompenses de plusieurs parties du robot (cont.)	22
3.14	Récompense Finale	22
4.1	Robot en cours de marche dans le simulateur Pybullet	25
4.2	Graphiques de récompense au cours de l'entraînement	26
4.3	Schéma d'un multi-layered perceptron (MLP) simple	29
4.4	Architecture du réseau de neurones de l'article [4]	29
4.5	Architecture du réseau de neurones personnalisée : utilisant une commande en vitesse en entrée	31

Table des figures

4.6	Architecture du réseau de neurones personnalisée : utilisant une commande en vitesse en entrée	32
4.7	Dérive du robot sur sa droite (Repère «world» au loin comme référence) . .	32
4.8	Courbe de reward pour un ré-entraînement avec fonction de reward modifiée et synchronisation de la référence désactivée	33
4.9	Contenu du fichier décrivant la marche d'un chien utilisée comme référence (38 frames et 19 paramètres)	34
4.10	HUFII dans le simulateur Matlab Simulink	35
4.11	Evolution du reward avec une référence incorrecte : chutes brutales très fréquentes dès le début de la simulation	36
4.12	Evolution du reward avec une référence corrigée et personnalisée	36

Chapitre 1

Contexte et présentation générale

1.1 Altran puis Capgemini Engineering

J'ai débuté mon stage au sein d'Altran Sud Ouest à Blagnac devenu ensuite Capgemini Engineering à la suite d'un rachat. Capgemini Engineering rassemble 52 000 ingénieurs et scientifiques dans près de 30 pays dont le rôle est d'accompagner les entreprises dans leurs projets de recherche et développement [1]. Ses services sont nombreux dans des secteurs tels que : l'aéronautique, l'automobile, le ferroviaire, les communications, l'énergie, les sciences de la vie, les semi-conducteurs, les logiciels et Internet, le spatial et la défense, et les biens de consommation. Comme le nom l'indique Capgemini Engineering est une "ligne d'activité globale" spécialisée R&D¹ appartenant au Groupe Capgemini, ESN² fournissant une expertise technique ou managériale pour les entreprises souhaitant "numériser" leur activité. Le groupe Capgemini, a été créé en 1967 par Serge Kampf à Grenoble sous le nom de Sogeti. Maintenant basée à Paris, la société fait partie du CAC40 à la bourse de Paris. [2]

1.2 Le département recherche

Comme expliqué précédemment, l'activité première de Capgemini Engineering est de fournir des experts pour répondre aux besoins techniques d'entreprises clientes. Cependant il existe un département indépendant de ce secteur : le département recherche. Il est organisé en plusieurs équipes développant chacune un projet de R&D. Le but de ces équipes est de construire un prototype pour faire une preuve de concept et potentiellement breveter certaines technologies mises au point. Pour financer le projet, l'équipe participe à des appels à projets et démarche de potentiels clients prêts à investir dans le futur produit. Les effectifs étant petits, l'équipe s'organise comme une startup. Chaque partie du projet (mécanique, électronique, vision par ordinateur, marche, logiciel) est traitée par une à trois personnes. Une telle organisation plus légère en termes de hiérarchie permet une bonne communication, circulation de l'information, donne plus d'autonomie à chacun, accélérant la production et la correction d'erreurs. J'ai effectué mon stage dans une de ces équipes : l'équipe du projet SUR³.

1.3 Mon projet et son équipe

1.3.1 Introduction

L'automatisation et la robotisation des lignes de production est devenu un enjeu majeur de l'ingénierie industrielle. Coûts de production réduits, vitesse de production augmentée et fiabilité des produits en sortie d'usine accrue, les entreprises d'aujourd'hui veulent avoir un contrôle total sur leurs moyens de production. Mon équipe essaye de

¹Recherche & Développement

²Entreprise de Services du Numérique

³Smart Universal Robot

répondre à cette demande en incluant à leur solution des technologies de pointe : machine learning, mobilité hybride à roue - quadrupède, prise de vue aérienne par drone.

1.3.2 L'équipe

L'équipe du projet SUR est composée à l'heure actuelle de 7 personnes réparties sur les différentes thématiques du projet.

Product Owner - Chef de Projet : Dr. CHENCHANA Bilel

Design mécanique : Dr. DIDI CHAOUI Mohammed

Design électronique : VAUBIEN Lionel

Génération de marche (modèle physique) : Dr. MOURAD Firas, JACOB Adrien

Génération de marche (machine learning) : OLIVARES David

Vision par ordinateur : Dr. LUCE-VEYRAC Pierre, KHAZEM Salim.

1.3.3 Tâches du robot

Les tâches du robot sont multiples : surveillance des équipements de la ligne de production (chaudières, fuites, températures d'équipements **se renseigner : peut être plus**) et l'assistance d'un opérateur humain pour la détection d'anomalies lors de l'assemblage des pièces d'un avion (boulon mal vissé, rivet au mal placé, défaut de peinture, câblages (**et plus ?**)). La plateforme robotique se doit d'être modulaire pour répondre à une grande variété de contraintes (inspection d'objets situés en hauteur, dans des espaces exigus, ou dangereux pour un opérateur humain). Devant ce vaste ensemble de tâches, créer une seule solution embarquant tous les outils pour faire face à toutes les situations serait contre productif. Un robot dimensionné et suréquipé de capteurs non nécessaire à un type de tâche, rendrait le robot plus lourd et énergivore. Notre projet propose donc un robot polyvalent et modulaire, composé d'une base mobile quadrupède et à roues, sur laquelle on peut ajouter différents modules et capteurs pour correspondre le plus possible à la tâche à accomplir sans pour autant encombrer le robot d'outils superflus. Le robot tire d'ailleurs son nom de cette approche hybride : **HUFII** Hybrid Unit For Industrial Inspection. Une solution annexe en cours d'étude est la collaboration du robot HUFII⁴ avec un drone quadrotor qui sera utile lors de l'inspection en hauteur de grandes installations. Pour l'instant, le moyen retenu pour la coordination des mouvements HUFII - drone est un système de câble (tethering system) reliant HUFII avec son drone et permettant aux deux plateformes de connaître leurs positions et vitesses par l'intermédiaire de la tension et orientation du câble.

⁴*Hybrid Unit For Industrial Inspection*

Chapitre 1. Contexte et présentation générale



FIG. 1.1 : Robot HUFII et ses modules interchangeables (de gauche à droite) : bras multi-fonctions, brase télescopique, drone (visualisation Blender)



FIG. 1.2 : Robot HUFII en contexte d'inspection industrielle (visualisation Blender)

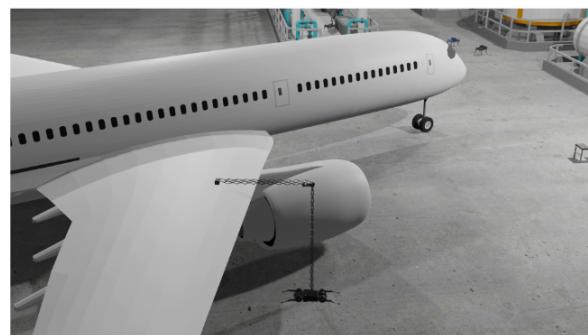


FIG. 1.3 : Robot HUFII, inspection peinture sur une aile d'avion (visualisation Blender)

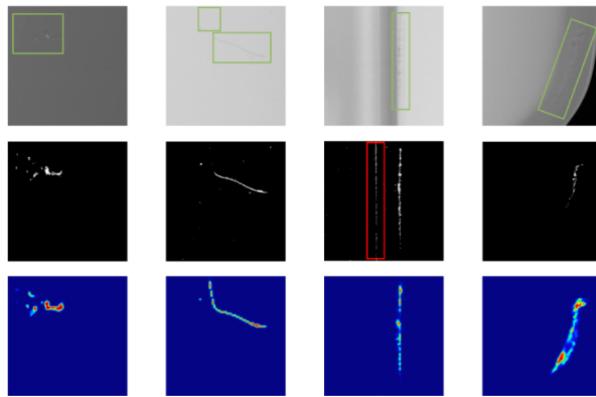


FIG. 1.4 : Exemple d’inspection, dommages de surfaces

1.3.4 Mobilité du robot

Il s’agit donc d’un robot mobile à roues et à pattes. Initialement le robot a été prévu pour se déplacer uniquement sur roues, mais l’environnement souvent accidenté d’une usine de production (câbles au sols, outils, échafaudages, escaliers, rampes etc.) nous oblige à rajouter un mode de locomotion supplémentaire : la marche quadrupède. Le robot possède donc quatre pattes, chacune composée de deux segments. Chaque patte est actionnée par trois moteurs : deux moteurs à la jonction entre le corps et la patte du robot (la hanche). Un moteur pour replier-tendre la patte et un autre pour bouger la patte latéralement, de gauche à droite.



FIG. 1.5 : Robot HUFII en mode quadrupède

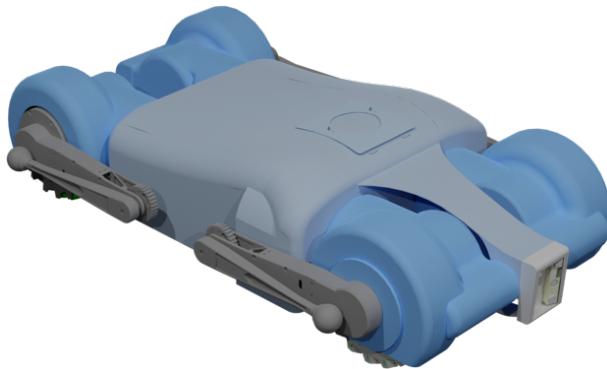


FIG. 1.6 : Robot HUFII en mode locomotion à roues

En ce qui concerne la mobilité à roues, le robot est équipée de quatre roues de type mécanum logées dans les épaules du robot en mode quadrupède. Ces roues viennent à toucher le sol lorsque le robot est en mode locomotion à roue, pattes repliées. Avec de telles roues le robot n'a pas besoin de roues directrices qui tournent, il peut se déplacer dans toutes les directions grâce aux roues mécanum. Le robot est dit "holonomique".

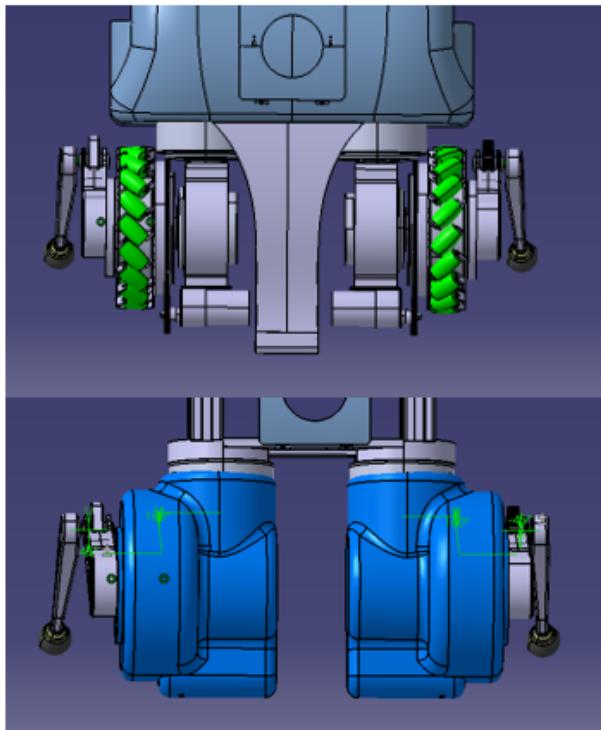


FIG. 1.7 : Roues mécanum dans l'épaule du robot HUFII

L'approche actuelle pour la génération de marche du robot, traitée plus en profondeur ultérieurement, est composée d'une partie modélisation cinématique du robot, automatique et cinématique inverse. L'autre partie, celle sur laquelle se concentre mon stage, concerne la marche robotique par des méthodes de machine learning, faisant appel notamment à une sous catégorie du ML⁵ : le Reinforcement Learning (RL).

⁵Machine Learning

1.3.5 La vision du robot

L'autre organe vital au robot est sa vision puisque c'est grâce à ses caméras qu'il effectuera la majorité des inspections et contrôles sur la ligne de production parmi lesquelles nous pouvons noter trois opérations principales : la détection de défauts, la détection d'objets et l'estimation de la pose (position et orientation) d'un objet dans l'espace. Pour pouvoir accomplir toutes ces tâches, la technologie commune utilisée est le machine learning (ML) et plus particulièrement le deep learning (DL). Ces deux sous ensembles de l'intelligence artificielle, se basent sur l'entraînement d'un réseau de neurones à partir d'une base de données, de modèles. Une fois le réseau entraîné on peut ainsi espérer que le réseau ait appris à classifier certains sous ensembles des images présentées, ici des défauts (rayures, impacts, corps étrangers etc.). L'autre objectif est d'identifier des pièces d'assemblage et d'estimer leur position et orientation dans l'espace, cela sert à la vérification d'un bon assemblage. Si la pièce est orientée de la mauvaise manière et pas au bon endroit dans l'assemblage final, le robot remontera l'anomalie à l'opérateur.

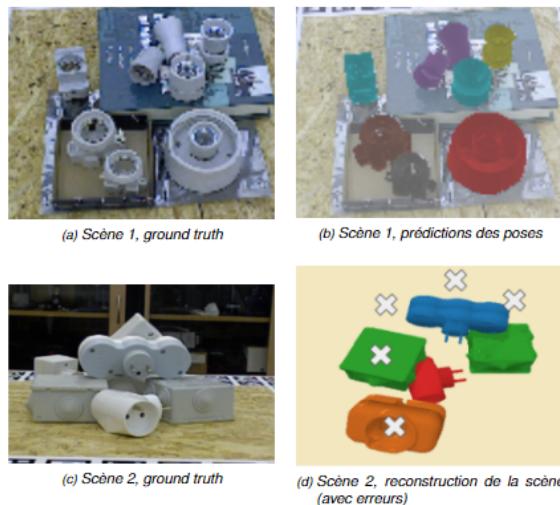


FIG. 1.8 : Exemples d'estimations de poses

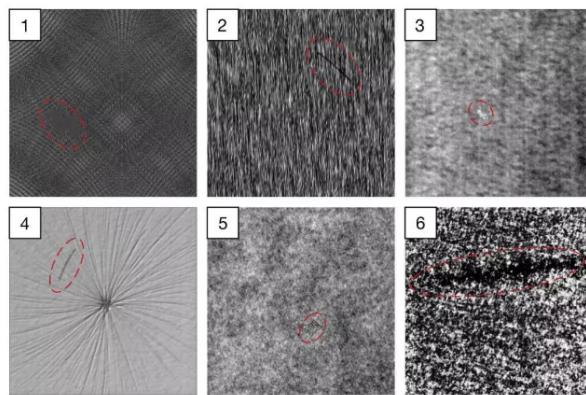


FIG. 1.9 : Exemple d'identification de défauts métalliques

1.3.6 Diagramme de Gantt

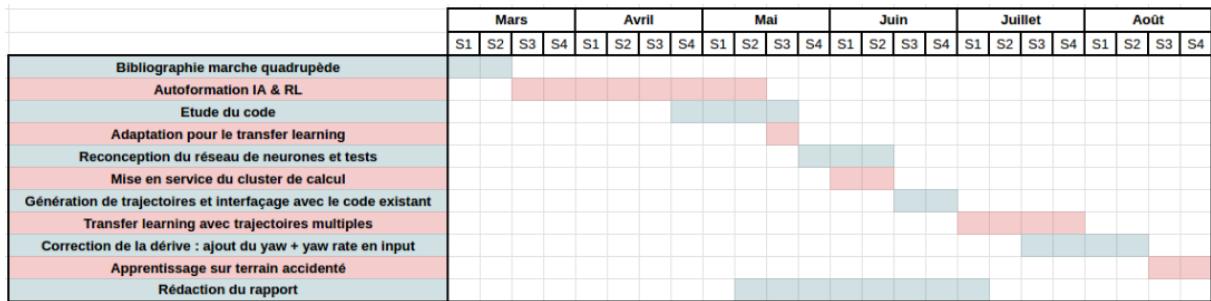


FIG. 1.10 : Diagramme de Gantt représentant la planification de mes tâches

Chapitre 2

Présentation de la politique de responsabilité sociale

Chapitre 3

État de l'art

3.1 Contexte et problématique

Pour accomplir la tâche de la marche quadrupède nous avons décidé d'explorer une nouvelle approche, mettant en avant une discipline qui a pris récemment en popularité : l'intelligence artificielle.

La raison de ce choix est la suivante : modéliser mathématiquement un robot quadrupède à 12 degrés de liberté parfaitement, ainsi que ses interactions avec son environnement peut se révéler être une tâche difficile voire impossible. Cela est dû à la grande complexité et instabilité inhérente d'un tel système mais aussi dû à l'infinité de paramètres, de perturbations, de phénomènes physiques rentrant en ligne de compte. [3]-[5]

On peut par exemple citer les différences de performances entre les actionneurs l'usure de ceux-ci ou des pièces du robot, les temps de calculs et de propagation de l'information entre les différentes couches logicielles etc.

De plus, une telle méthode nécessite de faire des approximations et de formuler des hypothèses pour que les modèles établis soient fonctionnels. Cela se traduit par des compromis sur les performances du robot : comme une accélération lente et/ou une vitesse des membres limitées. [5]

Dans l'objectif d'avoir un contrôleur robuste pouvant s'adapter à la majorité des situations, nous laissons donc le contrôle du robot à un ou plusieurs réseaux de neurones ayant appris par succession d'essais-échecs. Ces réseaux apprennent à contrôler le robot en utilisant des algorithmes d'apprentissage issus d'un sous-domaine du machine learning : l'apprentissage par renforcement (Reinforcement Learning¹ abrégé par RL). Ainsi les réseaux de neurones peuvent pallier les incertitudes non-modélisables, à généraliser les « connaissances » acquises afin de traverser des terrains plus ou moins obstrués et accidentés. [3]-[6]

Les concepts de machine learning et de reinforcement learning ne faisant pas partie de ma formation, il a fallu m'autoformer. Les premières parties de ce chapitre seront donc consacrées à une présentation de mes apprentissages dans le domaine. J'établirais ensuite un état de l'art pour finir par choisir un article comme base de mes travaux. Dans le chapitre suivant, je décrirai les réalisations et expériences menées chronologiquement en explicitant les changements d'approche et les difficultés rencontrées.

3.2 Le Reinforcement Learning

3.2.1 Généralités et Définitions

Le RL (Reinforcement Learning) est un sous-domaine du machine learning. Un concept important et fondateur du RL² est le MDP³ décrivant un système composé de [7] :

¹Dans ce rapport, par soucis de cohérence avec les appellations communément admises, le nom des algorithmes et méthodes seront nommées par leur nom anglais

²Reinforcement Learning (*Apprentissage par Renforcement*)

³Markov Decision Process (*Processus de Décision Markovien*)

- D'un agent autonome (ici un robot).
- D'un environnement (un simulateur pouvant générer différents types de terrains).
- D'un ensemble d'actions, noté A , que l'agent peut prendre (placer telle patte à tel endroit, transmettre un couple à un moteur d'une articulation).
- D'un ensemble d'états, noté S , qui représente la position et l'orientation du robot dans l'espace que l'on peut récupérer à partir des positions de toutes les articulations.
- Un ensemble de valeurs scalaires (une fonction) représentant la récompense, noté $R()$, par exemple positive lorsque l'agent a fait une bonne action, faible ou négative lorsqu'il a fait une mauvaise décision (score dans un jeu, distance parcourue par le robot, etc...).
- Une loi de probabilité représentant la loi de transition de l'agent, notée par la probabilité $P(s'|s, a)$ ou T , elle donne la probabilité qu'a l'agent de se retrouver à l'état s' s'il prend l'action a en partant de l'état s . Ainsi la probabilité $P(s'|s, a)$ contient l'incertitude que l'on a dans les actionneurs à atteindre notre consigne.
- Un scalaire γ , appelé discount factor, qui va donner moins d'importance aux actions et états futurs et plus d'importance aux actions et états proches dans le temps. Il est d'ordinaire compris entre 0.9 et 0.99.
- Une trajectoire, notée τ , qui n'est autre qu'une succession d'états au cours du temps que l'agent a pris.

Le robot est dans un état s , il prend l'action a , il arrive dans l'état s' avec la probabilité $P(s'|s, a)$ et se voit attribuer une récompense définie par la fonction $R()$.

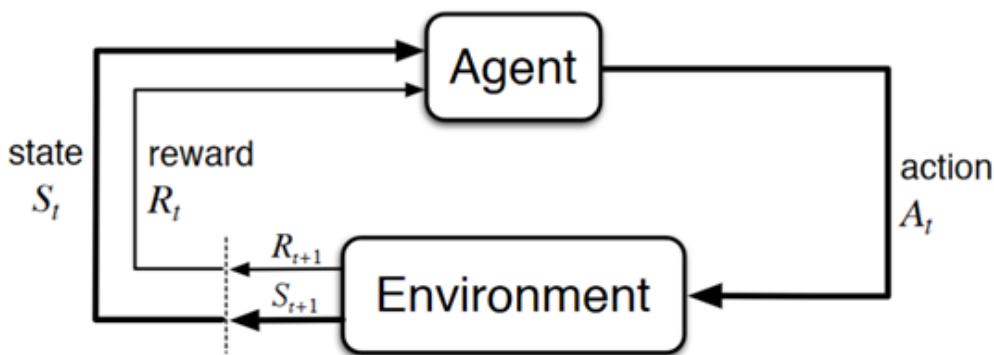


FIG. 3.1 : Schéma explicatif d'un Markov Decision Process

Maintenant que certains termes ont été définis, le but du RL est « d'apprendre » quoi faire – comment relier des états, situations à des actions – dans l'objectif de maximiser la récompense. [7]

L'agent ne sait pas quelle action il doit prendre, il découvre donc quelles actions lui donnent le plus de récompenses en les essayant. De plus, l'action prise n'affecte pas seulement la récompense immédiate, elle impacte l'état suivant et donc les récompenses à venir.

L'agent doit donc trouver une politique (policy notée π) optimale de contrôle de ses mouvements, c'est-à-dire une fonction qui retourne la meilleure action à prendre en fonction de l'état actuel. Par optimale, on entend que l'action retournée donnera la récompense maximale.

Beaucoup d'algorithmes résolvent cette problématique de trouver une politique (policy) optimale. Les algorithmes les moins récents se basent sur la construction d'une table répertoriant les couples (action, état) et en y associant une valeur appelée Q . On avait ainsi, pour chaque état, la liste des actions possibles et leurs valeurs respectives (somme espérée des récompenses futures).

$$V^\pi(s) = \sum_{s' \in S} [R(s, \pi(s), s') + \gamma V^\pi(s')]T(s, \pi(s), s'). \quad (3.1)$$

FIG. 3.2 : Equation de Bellman (Value Fonction) associant une valeur à chaque état du MDP

$$Q^\pi(s, a) = \sum_{s' \in S} [R(s, a, s') + \gamma V^\pi(s')]T(s, a, s'). \quad (3.2)$$

FIG. 3.3 : Equation de Bellman (Q-value) associant pour chaque couple (s,a) une valeur "Q" représentant la valeur d'une action à prendre dans un état donné.

En prenant l'action ayant la valeur la plus élevée, on était sûr de prendre l'action optimale.

Cependant, lorsque les ensembles d'états et d'actions sont trop grands, ces algorithmes deviennent très rapidement inefficaces. Par exemple, dans le cas d'un agent qui apprend à jouer à un jeu Atari, l'ensemble des états représenterait le nombre total de combinaisons possibles sur un écran (80x80 pixels même formaté en nuances de gris : $(256^{6400}$ combinaisons différentes). Stocker des valeurs et parcourir un tableau si grand est tout simplement impossible. C'est pourquoi on utilise un réseau de neurones qui va approximer cette table et donc ajouter du deep learning dans le processus. [8]

Un réseau de neurones est utilisé comme un approximateur de fonction (états de l'agent comme antécédent, action à effectuer comme image) paramétrisé avec les poids et biais (scalaires) notés θ des neurones. Au début de l'entraînement ces poids et biais sont initialisés aléatoirement. Au fur et à mesure que l'entraînement progresse, l'agent récolte des récompenses de la part de l'environnement. En quantifiant l'impact de chacun des neurones dans l'amélioration de la récompense obtenue (via un calcul de gradient), on met à jour les poids et biais du réseau dans le but d'augmenter la somme des récompenses obtenues par l'agent.

3.3 Principaux algorithmes existants

3.3.1 Policy Gradient (PG) algorithms

Cette catégorie d'algorithmes est présentée dans [9].

Dans l'état de l'art en contrôle robotique, on utilise le plus souvent une famille d'algorithmes RL appelés policy gradient algorithms [3]-[6], [10]. D'autres algorithmes existent mais sont soit destinés à être entraînés sans simulation [11], soit se basent sur un modèle préexistant décrivant le robot [12] (model-based RL). Dans les deux cas, ces alternatives sont moins récentes et moins nombreuses que les méthodes dites « model-free ». Ce sont des algorithmes utilisant un réseau de neurones pour modéliser la policy (notée π dont les poids sont généralement notés θ). Le réseau comporte typiquement des entrées correspondant à l'état de l'agent à un instant t et envoie en sortie l'action à effectuer. D'ordinaire, lors d'un entraînement supervisé les données d'entraînement sont correctement labelisées, ici on collecte les « labels » (actions à « attacher » à un état) pendant l'entraînement, au fil des interactions de l'agent avec le simulateur. Pour trouver la policy optimale, on fait interagir l'agent avec son environnement un certain nombre de fois et on recueille les trajectoires effectuées par l'agent. La somme des récompenses au cours d'un essai est calculée, puis on fait la moyenne de ces sommes par rapport au nombre d'essais (trajectoires) parcourues au cours d'une batterie d'essais. Cette fonction est aussi appelée objective function.

$$J(\theta) = E_{\pi\theta}[\sum \gamma r] \quad (3.3)$$

FIG. 3.4 : Score des essais noté J . Correspond à la moyenne des récompenses obtenues au cours des essais en suivant la policy π_θ

On calcule ensuite le gradient de ce score par rapport aux paramètres du réseau de neurones approximant la policy. Comme on veut maximiser le score de la policy, on utilise alors une méthode de gradient ascendant. On cherche ainsi à maximiser la probabilité des actions qui contribuent à une haute récompense tout en minimisant la probabilité des actions qui contribuent à une faible récompense [7], [12].

Lorsqu'on dérive $J(\theta)$ (appliquer l'opérateur gradient en fonction de la variable θ), on obtient :

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta(\log_\pi(s, a, \theta))R(\tau)] \quad (3.4)$$

FIG. 3.5 : Gradient de la fonction score (objective function) de la policy π

On utilise ce gradient dans un algorithme d'optimisation mathématique appelée : une montée de gradient (gradient ascent), afin trouver les poids θ du réseau permettant de maximiser le score de la policy :

Policy (représentée par le réseau de neurones) : π_θ
 Objective function : $J(\theta)$
 Gradient : $\nabla_\theta J(\theta)$
 Update : $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

FIG. 3.6 : Description de l'algorithme de gradient ascendant pour optimiser la policy

De cet algorithme dérivent plusieurs versions plus avancées mais gardant la même base et s'attaquant aux problématiques de temps d'entraînement et de convergence. Par exemple en réduisant la variance de l'estimée du score de la policy, en réduisant l'impact de mauvaises actions sur l'estimée de l'objective function et donc du gradient. Les algorithmes le plus souvent rencontrés dans le contrôle robotique et plus spécifiquement dans la marche robotique sont PPO [13] (Proximal Policy Optimization) et TRPO [14] (Trust Region Policy Optimization), tous deux variantes améliorées de l'algorithme policy gradient expliqué ci-dessus.

3.4 Améliorations du Policy Gradient

L'algorithme de RL le plus utilisé dans le contrôle robotique est PPO (Proximal Policy Gradient) [4], [6], [9], [13], [15]. Il est basé sur le Policy Gradient Algorithm et sur l'Advantage Actor Critic (A2C) [16].

3.4.1 Advantage Actor-Critic

L'algorithme A2C est décrit en détail dans le papier [16] et dans l'article web [17].

Rappelons l'expression de l'objective function du policy gradient classique :

$$J(\theta) = E_\pi[\log_\pi(\tau|\theta)R(\tau)] \quad (3.5)$$

A la place de la fonction de récompense $R(\tau)$, représentant la somme des récompenses perçues au cours des trajectoires τ on introduit la notion de fonction « avantage » (advantage function). Cette advantage function est plus efficace pour déterminer si l'action prise est plus intéressante :

$$A(s, a) = Q(s, a) - V(s) \quad (3.6)$$

FIG. 3.7 : Advantage Function soustrayant la Q-value du couple (état, action) avec la valeur moyenne de l'état

Cette fonction représente à quel point l'action choisie pour un état donné est « intéressante » par rapport à la moyenne des actions choisissables dans ce même état. Elle donne une indication plus intéressante sur la qualité d'une action que la fonction de récompense $R(\tau)$.

Si $A > 0$, l'action choisie (donnée par la sortie du réseau de la policy) est plus avantageuse que n'importe quelle autre action possible dans ce même état.

Cependant cette expression nous oblige à utiliser deux réseaux de neurones supplémentaires pour approximer la fonction $Q()$ et la fonction $V()$ utilisées dans le calcul de $A()$. Une nouvelle formulation de cette fonction $A()$ en utilisant la définition de la Q-function nous permet de contourner le problème :

$$\begin{aligned} A(s, a) &= Q(s, a) - V(s) \\ \text{or, } Q(s, a) &= r + \gamma V(s') \\ \text{donc : } A(s, a) &= r + \gamma V(s') - V(s) \end{aligned}$$

FIG. 3.8 : Réécriture de l'Advantage Function

Cette réécriture permet de manipuler un seul réseau uniquement pour approximer $V()$ et non plus $Q()$ et $V()$. Voici donc la fonction de score (objective function) avec $R(t)$ remplacée par $A(s, a)$:

$$J(\theta) = E_t[\log_{\pi\theta}(a_t|s_t)A_t] \quad (3.7)$$

Ainsi :

- Si $A(s,a) > 0$: notre gradient indiquera cette direction pour trouver le maximum de l'advantage function.
- Si $A(s,a) < 0$: notre gradient indiquera la direction opposée.

Un autre point fort de cet algorithme est qu'il est possible de le paralléliser l'entraînement sur plusieurs agents dans plusieurs environnements.

3.4.2 Proximal Policy Optimization (PPO)

Les articles [16][17] décrivent cet algorithme. Le PPO se base sur A2C qui lui-même se base sur le policy gradient.

Le défaut des techniques énoncées ci-dessus (Policy Gradient et A2C) sont qu'elles sont sensibles à la variabilité des données d'entraînement. A l'inverse de l'apprentissage supervisé, on entraîne notre agent grâce à un dataset qui se construit au fil des expériences de l'agent avec son environnement. Ce dataset d'entraînement n'est donc pas 100% parfait et statique mais s'améliore et devient de plus en plus correct à mesure que les expériences s'enchaînent. La distribution les actions et des récompenses est donc très variable, changeant à chaque update de la policy. Cette instabilité rend la tâche de trouver un maximum global à la fonction d'objectif difficile puisque le gradient peut mettre à jour le réseau avec de moins bon paramètres. L'objectif du PPO est donc de limiter la mise à jour des paramètres (poids et biais) du réseau π dans un certain intervalle et ainsi réduire l'instabilité d'entraînement.

Repartons de la fonction de score utilisée dans l'A2C :

$$J(\theta) = E_t[\log_{\pi\theta}(a_t|s_t)A_t] \quad (3.8)$$

A la place d'utiliser la probabilité $\log_{\pi\theta}(a_t|s_t)$ pour surveiller l'impact des actions, on peut utiliser le ratio entre la probabilité de l'action calculée avec le réseau $\pi\theta$ et calculée avec le réseau à la mise à jour précédente : $\pi_{\theta_{old}}$, noté $r_t(\theta)$:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \text{ so } r(\theta_{old}) = 1. \quad (3.9)$$

Si $r_t(\theta) > 1$, cela se traduit par : « on va prendre l'action a_t plus souvent avec cette policy »

Si $r_t(\theta) < 1$, cela se traduit par : « l'ancienne policy nous donne une action plus probable que la policy actuelle »

La nouvelle objective function, notée désormais L^{CPI} s'écrit donc :

$$L^{CPI}(\theta) = \widehat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\widehat{A}_t\right] = \widehat{E}_t[r_t(\theta)\widehat{A}_t] \quad (3.10)$$

Mais sans contrainte, une valeur de $r_t(\theta)$ trop importante (très inférieure ou très supérieure à 1) et le gradient de cette fonction serait trop important et entraînerait une mise à jour excessive de la policy. Nous allons donc pénaliser les changements qui s'éloignent trop de la précédente mise à jour.

Pour ce faire, avec PPO on prend le minimum entre L^{CPI} et la version « bornée ou clipped » de L^{CPI} entre $1-\epsilon$ et $1+\epsilon$. Voici donc la nouvelle objective function noté L^{CLIP} :

$$\begin{aligned} L^{CLIP}(\theta) &= \widehat{E}_t[min(r_t(\theta)\widehat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t)] \\ L^{CLIP}(\theta) &= \widehat{E}_t[min(L^{CPI}(\theta), clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t)] \end{aligned} \quad (3.11)$$

Ainsi les mises à jour des poids de la policy seront confinés à l'intervalle $[1-\epsilon, 1+\epsilon]$, avec ϵ une constante valant traditionnellement 0.2. [13]

On peut ensuite affiner cette erreur en ajoutant deux termes supplémentaires : $L_t^{VF}(\theta) = (V_\theta(s_t) - V_t^{target})^2$ et $S[\pi_\theta](s_t)$. L_t^{VF} est une erreur qui permet de mettre à jour la partie value function du réseau et $S[\pi_\theta](s_t)$ est une mesure d'entropie de la distribution des actions donnée par la policy π_θ , utilisée ici comme bonus (+) pour favoriser l'exploration de nouveaux états par la policy.

$$L^{CLIP+VF+S}(\theta) = \widehat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \quad (3.12)$$

3.5 Apprentissage et récompenses

Lors des précédentes parties nous avons vu que le choix d'un bon algorithme (choix d'une bonne objective function) était très important. Mais il est tout aussi, voire plus important de bien concevoir la fonction de récompense (notée $R()$) puisque c'est elle qui

va déterminer la valeur de l'état dans lequel on se trouve (par exemple se trouver les 4 pattes en l'air doit être un état avec une valeur faible et inversement être droit sur ses pattes, bien stable, doit être un état avec une valeur importante). Bien sûr, ces deux exemples sont des extrêmes, il existe une infinité d'états possibles chacun ayant sa propre valeur $V(s)$ (valeur de l'état s). Dans le cadre de l'apprentissage de la marche on peut trivialement créer une fonction de récompense formulée ainsi : « plus l'agent avance loin (selon l'axe x par exemple), plus il gagne des points (des récompenses) »

$$R(t) = \sqrt{x(t)^2} \quad (3.13)$$

Cependant, avec une telle fonction de récompense, nous verrons très certainement l'agent adopter des comportements qui vont certes le faire avancer et donc gagner des récompenses mais d'une manière inattendue. Par exemple, avec cette seule contrainte sur la récompense, on peut tout à fait imaginer que l'agent pourra trouver une manière de se déplacer en... rampant ce qui n'est pas le résultat voulu. De manière générale, essayer de trouver manuellement la fonction de récompense parfaite pour que l'agent apprenne à marcher exactement comme on veut est rarement une approche concluante. En effet, on ne peut pas absolument prévoir tous les comportements sous-optimaux d'un agent pour les pénaliser au sein de sa fonction de récompense.

3.6 Veille scientifique

Il existe beaucoup de littérature scientifique sur le sujet de la marche robotique, du contrôle robotique via machine learning et un peu moins sur la combinaison de ces deux disciplines. Mais voici un récapitulatif des différents papiers notables parmi lesquels j'ai choisi celui sur lequel j'allais baser mon travail.

[11] est un papier se concentrant sur la rapidité et l'efficacité en termes de nombres d'essais. Pour ce faire, les chercheurs proposent une extension d'un algorithme appelé Soft Actor Critic (SAC), lui-même basé sur l'A2C présenté précédemment en 3.4.1. Cependant, l'équipe ne se concentre que sur de l'apprentissage hors simulation résultant une démarche peu esthétique. L'entraînement de l'agent en simulation est quelque chose de commun à tous les papiers présentés ci-après. Les agents entraînés en simulation présentent des allures tout à fait fluides pour la plupart comparativement à leurs homologues entraînés hors simulation. De plus, lorsque l'entraînement se déroule en simulation, on peut paralléliser l'entraînement sur plusieurs environnements pour accélérer le processus et réduire drastiquement les délais. La simulation est donc un avantage conséquent dans l'entraînement d'agents contrôlant la marche d'un robot. Son seul désavantage est qu'il n'existe malheureusement pas de simulateur parfait, réaliste à 100%, dont la simulation équivaille au monde réel.

[5] est un papier complet, découpant l'apprentissage sur plusieurs réseaux de neurones pour répondre à plusieurs problématiques. Du fait qu'aucun simulateur ne reproduit à 100% la réalité, le réseau entraîné sur simulateur a beaucoup de mal à atteindre les mêmes performances dans la réalité, c'est ce que l'on appelle le *reality gap*. Pour absorber cet écart entre simulation et réalité, l'équipe crée et entraîne un réseau supplémentaire

tenant en entrée une position et donnant un couple moteur en sortie. Ce réseau n'utilise pas d'algorithme de RL mais apprend avec des techniques plus communes de DL⁴.

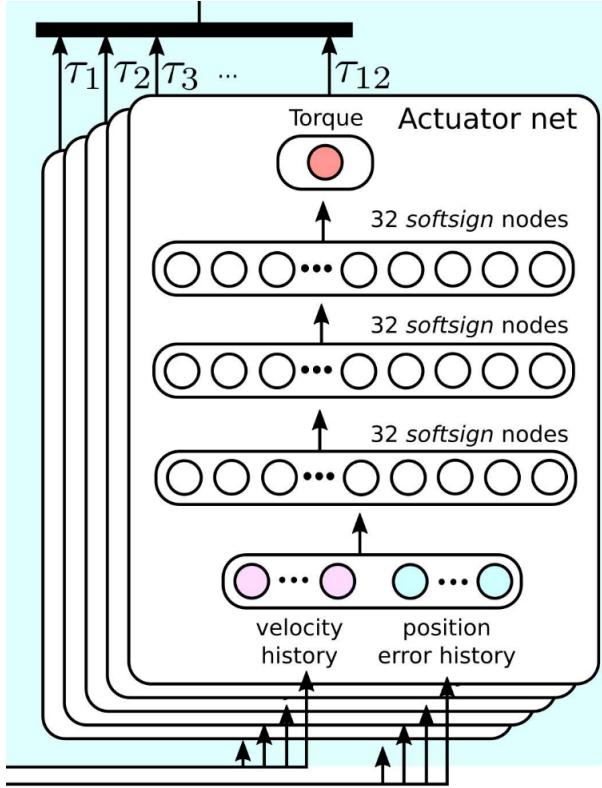


FIG. 3.9 : Réseau absorbant le reality gap (Actuator Net)

Pour le réseau de la policy, les chercheurs séparent l'apprentissage en deux phases. D'abord on entraîne un réseau maître, ayant accès en entrée à des informations privilégiées (connaissance parfaite du terrain, des obstacles et des forces de contact) en plus de l'état du robot (position et orientation du corps et des membres du robot). Cette phase d'apprentissage se fait avec du RL, puisque qu'il n'y a aucune base de données à partir de laquelle apprendre. Il y a ensuite une deuxième phase au cours de laquelle on entraîne un second réseau (élève) par de l'apprentissage supervisé, grâce aux informations de sortie du réseau maître. Ce réseau élève est un TCN (Temporal Convolution Network) spécialement conçu pour prendre, en entrée, un historique temporel d'états précédent du robot. Cette architecture se contraste des méthodes plus classiques ne se basant que sur un instantané (un seul état du robot à l'instant t) avant d'émettre une action à effectuer. De plus, ce réseau élève est entraîné cette fois ci sans les informations privilégiées auparavant accessibles. C'est à dire qu'il ne dispose plus que des informations que lui renvoient ses capteurs proprioceptifs (absence de caméra/vision) : encodeurs aux articulations et centrale inertie.

⁴Deep Learning

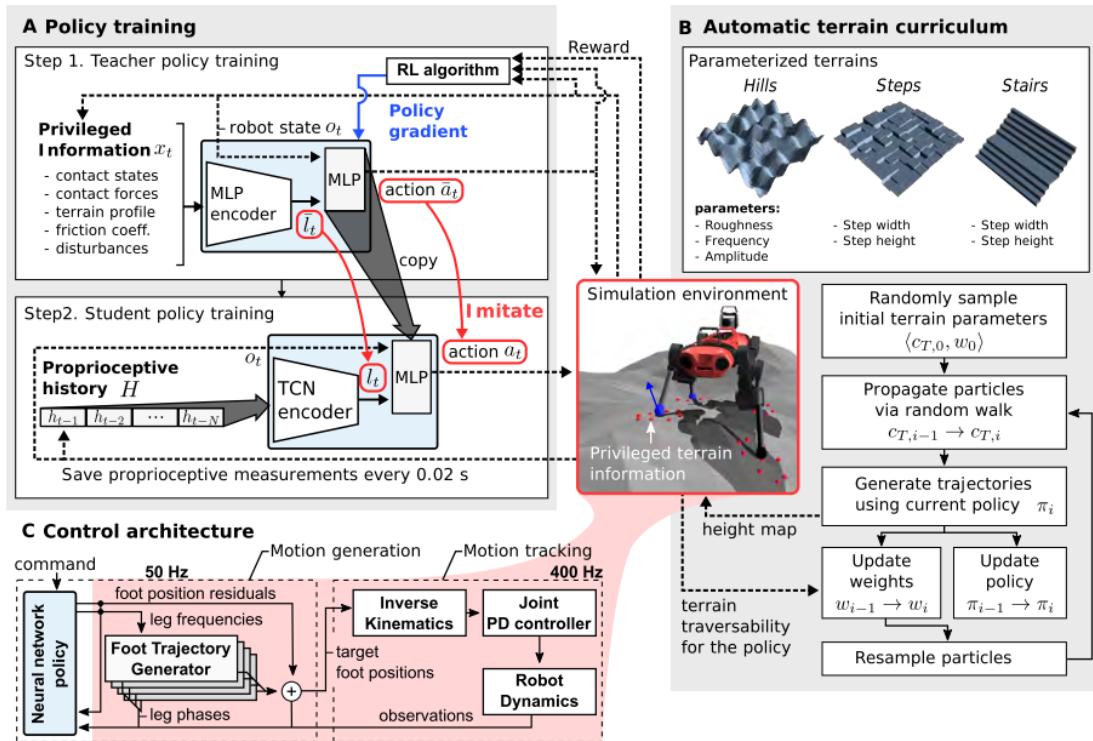


Fig. 4. Overview of the presented approach. (A) Two-stage training process. First, a teacher policy is trained using reinforcement learning in simulation. It has access to privileged information that is not available in the real world. Next, a proprioceptive student policy learns by imitating the teacher. The student policy acts on a stream of proprioceptive sensory input and does not use privileged information. (B) An adaptive terrain curriculum synthesizes terrains at an appropriate level of difficulty during the course of training. Particle filtering is used to maintain a distribution of terrain parameters that are challenging but traversable by the policy. (C) Architecture of the locomotion controller. The learned proprioceptive policy modulates motion primitives via kinematic residuals. An empirical model of the joint PD controller facilitates deployment on physical machines.

FIG. 3.10 : Schéma architectural Master - Student

Grâce à toutes ces contributions, les chercheurs de cette équipe arrivent à des résultats impressionnantes en conditions réelles. Le robot se déplace efficacement sur des terrains variés, herbe, terre, graviers, eaux peu profondes, il gravit des pentes, et arrive même à monter une marche d'escalier. Le seul bémol de ce papier, est la grande complexité des méthodes mises en place, le simulateur propriétaire utilisé, et la non-disponibilité d'un code source sur lequel se baser.

[6] base son approche également sur une division en deux réseaux un planificateur et un contrôleur. Le planificateur génère une combinaison de pas à faire en fonction du terrain présenté au robot. Il se charge de déterminer la faisabilité d'un pas (vide à éviter, pente à compenser, patte à poser trop éloignée du corps, donc risque de déstabiliser le robot...). Il relie la position actuelle du robot à sa destination par une séquence de pas faisables. Le deuxième réseau, le contrôleur exécute la séquence en restant le plus fidèle possible à la trajectoire des pas proposé par le planificateur. Il maintient l'équilibre et gère les perturbations.

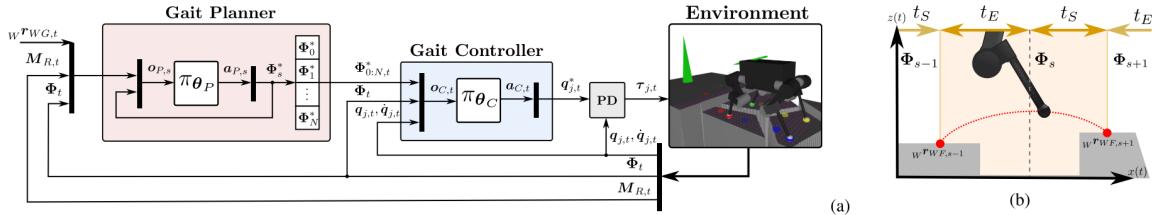


Fig. 2: (a) Overview of the proposed control structure used at deployment time. (b) Phases within a sequence are indexed using s , and every index corresponds to a point in time centered around a window defined by the durations t_E and t_S . The center of the window is defined by the motion of the base as captured by the phase Φ_s . t_S defines the time-to-switch from the current contact support to the next, specified in Φ_{s+1} , and t_E defines the time elapsed since the switch from the previous contact support, specified in Φ_{s-1} , to the current.

FIG. 3.11 : Architecture globale : planner et controller

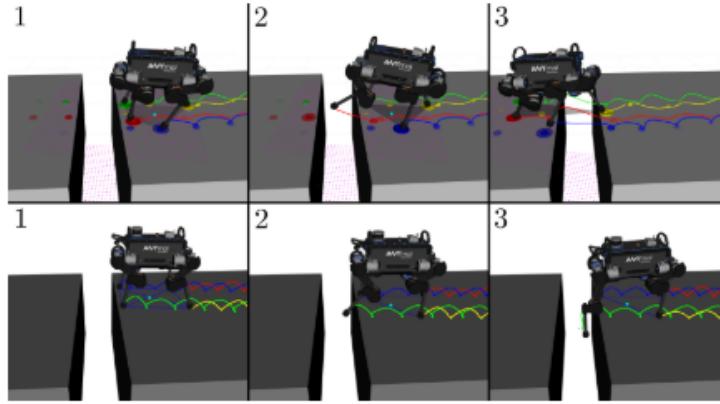


Fig. 5: Snap-shots of the comparison between our policies (top row) and Free-Gait (bottom row) overcoming the 40 cm gap. The figures are numbered left-to-right to indicate the order of the frames.

FIG. 3.12 : Démonstration en simulation (pas : ronds sur le sol, lignes : trajectoires)

[4] Pour contraindre notre agent à adopter des comportements sûrs et optimaux, on peut introduire dans la fonction de récompense un ou des termes récompensant les mouvements de l'agent proches des mouvements « idéaux ». Dans ce papier de recherche [4], les auteurs utilisent une motion capture d'un vrai chien ayant plusieurs démarches (trot, pas, sauts, tourner autour de soi-même, tourner en sautant, pas chassés) pour contraindre leur agent. La fonction de récompense prend donc simplement la forme d'une somme de termes où chacun des termes représente la différence entre la position et la vitesse de la motion capture de chien et le robot quadrupède en apprentissage :

$$r_t^p = \exp \left[-5 \sum_j \|\hat{q}_t^j - q_t^j\|^2 \right] \quad r_t^v = \exp \left[-0.1 \sum_j \|\hat{q}_t^j - \dot{q}_t^j\|^2 \right]$$

(a) proximité entre les positions des articulations (motion cap et agent)

$$r_t^e = \exp \left[-40 \sum_e \|\hat{x}_t^e - x_t^e\|^2 \right] \quad r_t^{rp} = \exp \left[-20\|\hat{x}_t^{root} - x_t^{root}\|^2 - 10\|\hat{q}_t^{root} - q_t^{root}\|^2 \right]$$

(c) proximité entre les positions des pattes (motion cap et agent)

(d) proximité entre les positions et orientations du corps (motion cap et agent)

FIG. 3.13 : Sous-récompenses de plusieurs parties du robot

$$r_t^{rv} = \exp \left[-2\|\hat{x}_t^{root} - \dot{x}_t^{root}\|^2 - 0.2\|\hat{q}_t^{root} - \dot{q}_t^{root}\|^2 \right]$$

(e) proximité entre les vitesses linéaires et angulaires du corps (motion cap et agent)

FIG. 3.13 : Sous-récompenses de plusieurs parties du robot (cont.)

La récompense finale est une somme pondérée des sous-récompenses, chaque sous-récompense étant pondérée par un poids propre à elle, dans l'objectif d'accorder plus ou moins d'importance à chaque terme dans la somme finale.

$$r_t = w^p r_t^p + w^v r_t^v + w^e r_t^e + w^{rp} r_t^{rp} + w^{rv} r_t^{rv}$$

(a) Récompense finale : Somme pondérée des sous-récompenses

$$w^p = 0.5, w^v = 0.05, w^e = 0.2, w^{rp} = 0.15, w^{rv} = 0.1$$

(b) Poids des sous-récompenses

FIG. 3.14 : Récompense Finale

Ce papier gère aussi à sa manière le reality gap en prévision d'un export du réseau policy vers une plateforme physique. L'équipe s'appuie sur des solutions déjà connues pour réduire l'impact du reality gap [3]-[5], [18]. L'objectif est d'aléatoiriser les caractéristiques physiques du robot au cours de l'entraînement. Par exemple, toutes les 1000 itérations l'algorithme change le poids, la longueur des membres, la gravité du simulateur, les coefficients de friction. Pour approfondir sur cette idée, les chercheurs peuvent fournir au cours de l'entraînement, les paramètres dynamiques (notés μ par exemple) en entrée du

réseau policy. Ils passent d'une policy de cette forme : $\pi(a|s)$ à $\pi(a|s, \mu)$. Ils obtiennent une policy paramétrisée par μ , appelée « strategy » dans la littérature [19]. Il s'agit ensuite de trouver les paramètres optimaux μ^* correspondant au monde réel. Pour cela on procède par batterie d'essais (*batch of rollouts*) sur le robot réel. A chaque début d'essai on sélectionne aléatoirement selon une distribution gaussienne des paramètres μ . On instancie une strategy que l'on exécute sur le robot réel et on mesure ses performances (son score, même chose que l'objective function dans les parties précédentes). A chaque itération, les matrices de moyenne et de covariance de la distribution gaussienne utilisée pour choisir les paramètres μ sont mises à jour pour augmenter le score de la strategy lors de la prochaine itération [20].

Dans l'exemple du robot imitateur [4], les paramètres μ ne pouvant potentiellement pas contenir toutes les caractéristiques physiques d'un environnement, les auteurs ont ajouté un « information bottleneck » pour induire de cette manière une incertitude, un biais dans la modélisation de μ dans un réseau de neurones. De cette façon, lors de la mise à jour des *strategy* par d'autres plus robustes, les auteurs prennent ainsi en compte le caractère non-définissable et incomplet de μ et aboutissent à des *strategy* plus optimales.

J'ai donc choisi ce papier [4] en tant que base de mes travaux, le code fourni par l'équipe en libre accès me permet d'avoir un framework existant, avec un simulateur : Pybullet dans lequel travailler. Le papier utilise l'algorithme le plus récent et performant du Reinforcement Learning : PPO⁵. L'objectif à long terme est de fusionner les solutions de cet article avec certaines autres, issues des articles mentionnés auparavant dans l'état de l'art.

⁵Proximal Policy Optimization (*Optimisation de la Politique Proximale*)

Chapitre 4

Travaux et réalisations

4.1 RéPLICATION DES RÉSULTATS ET LIMITES DU PAPIER CHOISI

Lors des tests nous avons pu constater que l'agent aboutissait à une démarche tout à fait acceptable en simulation. Le robot marche droit vers les x positifs, présentant cependant une petite dérive vers la droite (axe y négatifs).

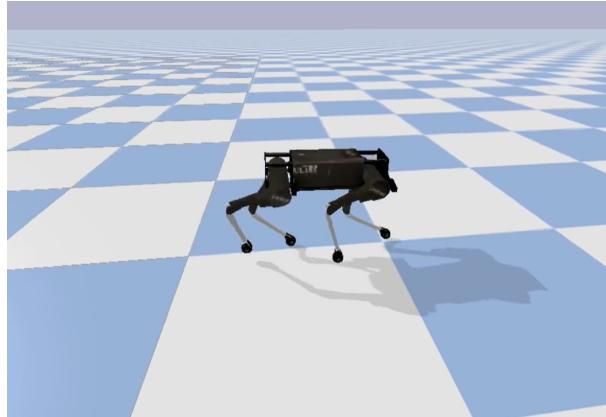


FIG. 4.1 : Robot en cours de marche dans le simulateur Pybullet

Cependant, ces résultats ne sont pas exempts de défauts. Premièrement, comme le titre de l'article l'indique, ces travaux ont été mis au point pour imiter la marche d'un chien. Dans son architecture le réseau prend comme entrées un «but» (une liste de positions d'articulations issues de la référence du chien). Cela pose problème lors de l'exploitation du réseau après son entraînement. Nous aurions besoin de commander le robot (le réseau) en lui donnant des vitesses (x, y, z, r, p, y) et non en lui fournissant des positions articulaires à rejoindre. Pour cela nous pensions qu'il était suffisant de ré-entraîner le réseau imitateur tel quel avec des commandes personnalisées. Le but étant de se servir des connaissances en marche du réseau imitateur pour y ajouter une sur-couche en le ré-entraînant (transfer learning¹).

Une autre limite jumelle concernant la commandabilité du robot est la simplicité de la trajectoire du robot. Le modèle que l'agent essaye d'imiter est un modèle de chien se déplaçant sur le long de l'axe x avec une vitesse d'environ 1.15 m.s^{-1} . Si l'agent n'est entraîné qu'avec une trajectoire en ligne il est impossible de le commander sur des trajectoires différentes (virages, ralentissements, accélérations). Un objectif supplémentaire est donc aussi, d'apprendre au robot à suivre plusieurs types de trajectoires, adopter des démarches sensiblement différentes pour suivre une commande variée.

L'avantage d'une architecture quadrupède étant le franchissement d'obstacles, le fait de pouvoir ré-exploiter un réseau capable de marcher sur terrain plat pour lui apprendre à marcher sur un terrain accidenté est très utile voire indispensable. Nous sommes partis du principe que la marche sur terrain plat est similaire à la marche sur terrain accidenté, puis nous enrichissons le réseau existant en l'entraînant sur des terrains différents.

Ce point et les précédents ont en commun l'usage du transfer learning. Cette technique de machine learning amène une dernière problématique. En effet, les auteurs ne

¹Méthode de machine learning consistant à ré-entraîner un réseau déjà entraîné pour qu'il apprenne de nouveaux comportements tout en gardant les anciennes connaissances comme base.

sauvegardent et ne chargent pas les hyper-paramètres de l'algorithme utilisé (PPO), mais uniquement les poids du réseau de neurones. Cette approche est suffisante pour tester un réseau après apprentissage, mais ne permet pas de reprendre un entraînement «là où l'on s'est arrêté» à proprement parler. La capacité de pouvoir poursuivre un entraînement est aussi un atout non négligeable dans le développement. Les entraînements étant très longs (dizaine de jours, hors utilisation du serveur de calcul) il est donc intéressant de pouvoir stopper un entraînement pour le poursuivre plus tard.

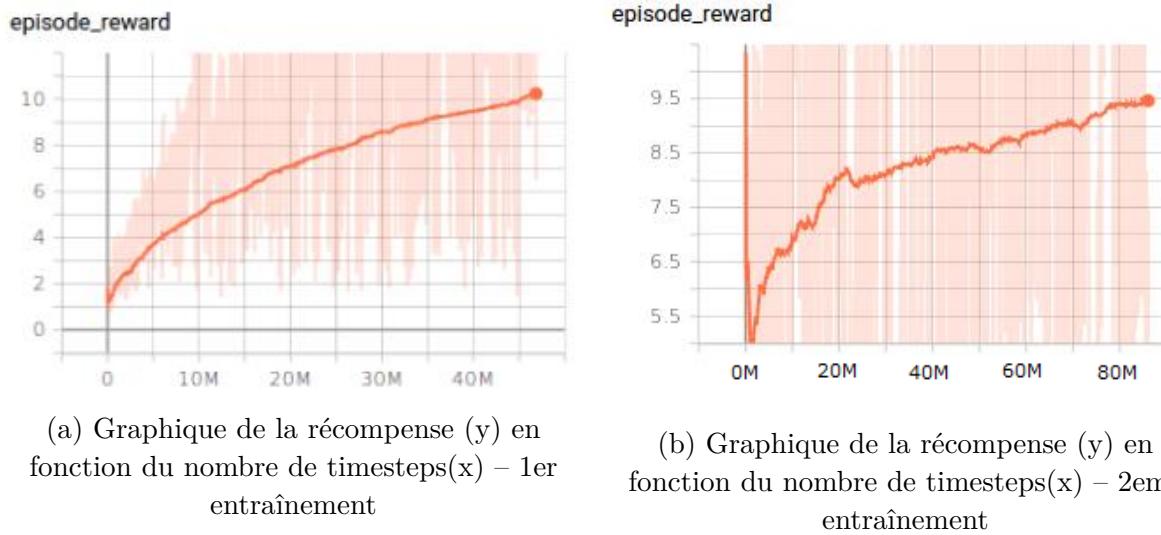


FIG. 4.2 : Graphiques de récompense au cours de l'entraînement

Les figures 4.2a et 4.2b ci-dessus illustrent bien le problème : la métrique mesurée ici est la récompense obtenue par l'agent. Il est logique qu'elle augmente progressivement (l'agent apprend à gagner de plus en plus de récompenses). En revanche, on constate qu'après une phase de grandes perturbations, au début du 2eme entraînement, on repart d'une récompense à peu près égale à 5, alors que nous avions stoppé le précédent entraînement à 10. Mais plus important encore, nous voyons qu'en 80 millions de *timesteps*², le réseau n'arrive pas encore à égaler les performances d'un agent entraîné une première fois en $\approx 50M$ de *timesteps*.

L'objectif est donc de résoudre ces trois limitations inhérentes au papier et au code fourni, puis d'éventuellement inclure d'autres algorithmes et concepts explicités dans la partie 3.6 notamment en ce qui concerne le franchissement d'obstacles [3], [5], [6].

4.2 Environnement et outils

L'environnement utilisé pour l'entraînement de l'agent est composé de plusieurs outils logiciels. Tout d'abord le système d'exploitation : GNU/Linux Ubuntu 16.04, il s'agit d'une version plutôt ancienne d'Ubuntu qui n'est plus supportée, mais compatible avec les travaux déjà existants de mes collègues développés sur ROS Kinetic Kame, nécessitant la version 16.04 d'Ubuntu. Très connu dans le milieu du développement, cette distribution

²Pas de temps

de GNU/Linux, facilite grandement l'installation de paquets/logiciels utiles comme des langages de programmation, librairies, simulateurs, versioning, etc.

Pour accélérer l'entraînement de mes modèles, j'ai également utilisé un serveur de calcul CPU. Cette ressource n'a été utilisable que 4 mois après le début de mon stage et s'est révélée être très utile en réduisant les temps d'entraînement (12 jours à 2 jours). Pendant les 4 premiers mois, j'ai effectué tous mes entraînements sur une machine locale à 8 coeurs et je suis passé sur 32 coeurs sur le serveur CPU.

Le langage utilisé est Python (ver 3.7.10). Ce langage est le plus commun dans le monde du machine learning, c'est un langage interprété, à la syntaxe peu typée et facile à prendre en main. Python possède une multitude de librairies installables facilement pour correspondre à tous besoins les plus spécifiques d'un projet.

Les librairies utiles et spécifiques à ma tâche sont :

- Stable Baselines 2 [21] : SB2³ est un fork de la librairie de OpenAI nommée «Baselines». Cette dernière implémente l'essentiel des algorithmes les plus connus et utilisés dans le monde du RL ainsi qu'un interfaçage avec un autre outil OpenAI décrit ci-après : Gym. Le fork Stable Baselines est un projet git open source maintenu par divers chercheurs en RL, il se veut être une amélioration de la librairie OpenAI Baselines. En effet, les auteurs de SB2 parlent d'une implémentation plus complète (plus d'algorithmes disponibles), mieux documentée et mieux testée que Baselines. En ce qui concerne le *backend* (architecture du réseau, types de couches, optimiseurs, outils de debug), SB2 s'appuie sur le framework Tensorflow 1.15. Cette version étant peu à peu dépassée par d'autres frameworks plus récents (Tensorflow 2.x, Pytorch), SB2 a évolué très récemment vers une troisième version cette fois-ci utilisant Pytorch pour la partie backend.
- OpenAI Gym [22] : Gym est une boîte à outils pour développer et tester des algorithmes d'apprentissage par renforcement. Il permet d'enseigner aux agents toute sorte de tâches, de la marche aux jeux comme Pong ou Pinball. C'est le framework qui donne une structure à notre environnement d'apprentissage. Un environnement Gym, consiste en une interface de programmation permettant de décrire un MDP :
 - Espaces d'action et leurs types (piston d'amplitude A , moteur de plage de rotation entre $[-\pi, \pi]$, se déplacer d'une ou plusieurs case sur un tableau de jeu, déplacer le palet d'un pixel vers la droite pour un jeu de casse-brique par exemple, etc.).
 - Espaces d'état et leurs types (encodeur sur un/des moteurs donc un flottant ou une liste de flottants entre $[-\pi, \pi]$, vitesses linéaires ou angulaires donc un 6-tuple de flottants, des coordonnées sur un tableau de jeu, état de l'écran pour un jeu vidéo).
 - La fonction de récompense qui définit l'objectif de l'agent.
 - Une fonction pour agir sur l'agent, pour "step" d'un pas de temps dans l'environnement.

³Stable Baselines 2, librairie python pour le Reinforcement Learning

Gym sert de couche intermédiaire permettant la standardisation des informations circulant entre le simulateur Pybullet et la librairie de RL : Stable Baselines.

- Pybullet [23] : Pour faire évoluer l'agent sur son terrain nous avons besoin d'un simulateur. Pour ce faire le papier que j'utilise comme base de mes travaux utilise Pybullet. Comme son nom l'indique il s'agit d'une librairie python permettant d'importer la structure physique de notre agent et de créer un terrain. Il permet, via une interface graphique, de visualiser la simulation en temps réel pour tester l'entraînement d'un agent par exemple. A l'inverse lors de la phase d'apprentissage, on peut accélérer ce dernier en désactivant l'affichage de l'interface graphique.

4.3 Réalisations

4.3.1 Adaptation au Transfer Learning

La première étape fut d'adapter le code pour permettre la reprise d'un apprentissage après arrêt. Comme expliqué, cette fonctionnalité est très utile pour ré-entraîner un réseau existant afin d'apprendre des mouvements plus complexes en partant d'une marche «simple».

Après une étape de compréhension du code, j'ai trouvé la fonction utilisée pour la sauvegarde du modèle : `save_parameters()`. Lors de recherches dans la documentation de Stable Baselines, il s'est avéré qu'elle ne sauvegarde que les poids du réseau de neurones et non pas les hyper-paramètres d'entraînement, ni les paramètres de l'optimiseur (learning rate...) et de l'algorithme PPO (entropie, ϵ ...). Choix questionable de la part de l'équipe de l'article, puisqu'il existe une fonction `save()` disponible, qui sauvegarde une partie de ces hyper-paramètres.

***** INSERER FIGURE DE L'APPRENTISSAGE idéalement imitation + suite *****

Même si on observe qu'on reprend à un reward quasiment nul, on peut voir qu'en l'espace de 40M pas de temps (timesteps) le ré-entraînement atteint un niveau de reward égal à la valeur du réseau lorsqu'on a arrêté l'entraînement une première fois. Selon les forums de Stable Baselines, un des améliorations de la version 3 permettrait de sauvegarder l'intégralité des paramètres et ainsi de reprendre rigoureusement l'entraînement.

4.3.2 Rendre le réseau pilotable

Deuxièmement, comme expliqué en 4.1 le but premier est de rendre le réseau «pilotable» en vitesse.

Voici un schéma du réseau hérité du papier. Il se présente comme un multi-layered perceptron : un réseau où couches de neurones sont reliées dans un seul sens (input vers output) et chaque neurone reliant tous les neurones de la couche successive (couche dite *dense*).

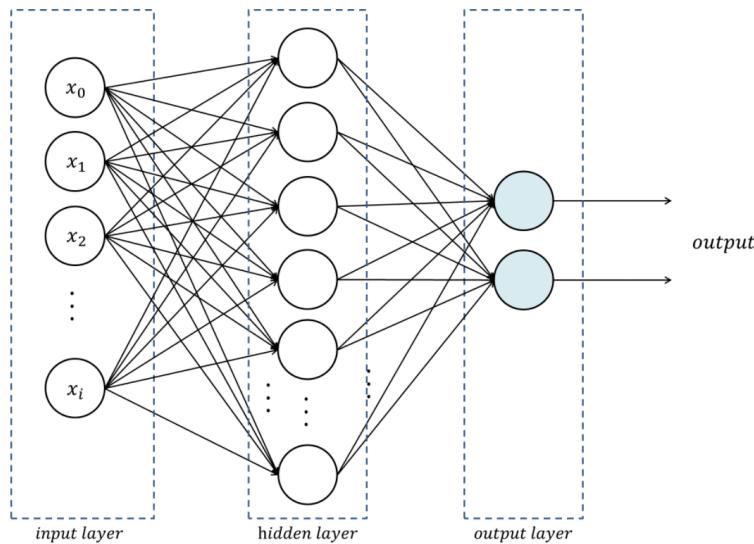


FIG. 4.3 : Schéma d'un multi-layered perceptron (MLP) simple

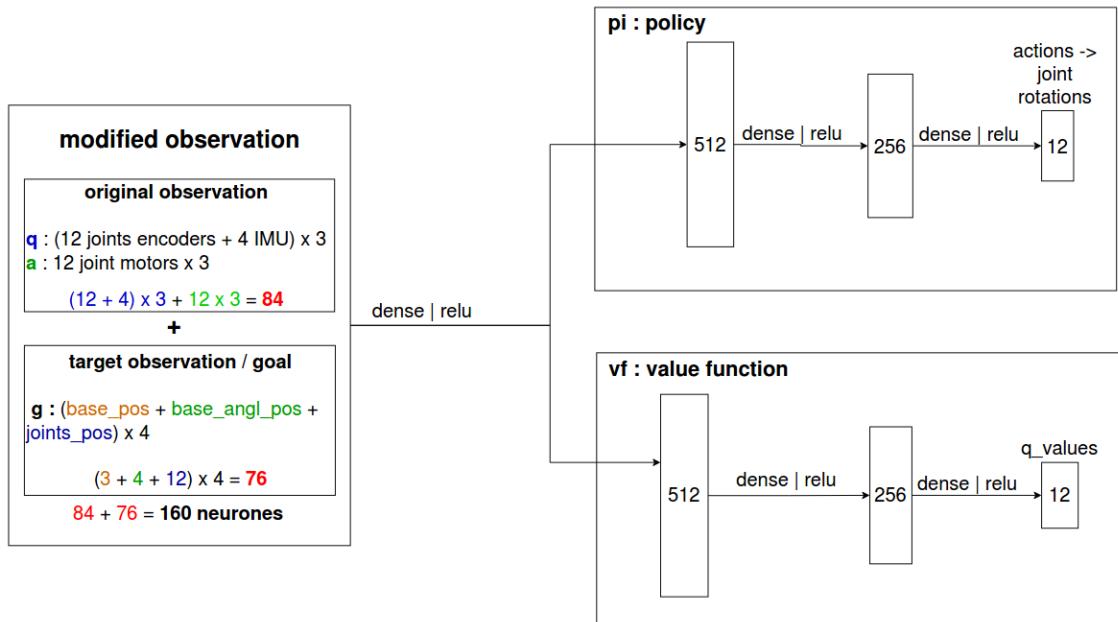


FIG. 4.4 : Architecture du réseau de neurones de l'article [4]

Sur figure ci dessus 4.4, on voit que la couche d'entrée (*input*) du réseau correspond à ce qu'on appelle une modified observation. Cette couche est composée de 2 types d'informations :

- L'observation originale (*original observation*) correspond à :
 - L'état, l'observation (au sens odométrique) du système (du robot) composée des 12 valeurs des positions angulaires des 12 articulations du robot (encodeurs) + 4 informations issues d'une centrale inertuelle (IMU) : position (roll, pitch) et accélération angulaires (roll rate, pitch rate) du corps du robot. Cette observation est récupérée sur trois fois aux temps ($t, t - 1, t - 2$), et ainsi fourni une information sur l'historique des états précédent du robot.

- À ceci s'ajoute un autre historique, celui de 3 actions passées ($t-1, t-2, t-3$). Ces deux historiques (états, actions) ont pour objectif d'entraîner un réseau qui prenne en compte le passé du robot (petit échantillon) pour prendre une décision sur l'action à prendre plus fiable et robuste.

On obtient ainsi 84 valeurs pour décrire toutes ces informations, donc 84 neurones d'input.

- L'observation cible ou «but» (target observation, goal). Elle représente la référence, la motion capture du chien qui marche. On donne au réseau 4 buts aux temps ($t+1, t+2, t+10, t+30$). Elle est composée de : la position du corps du robot (x, y, z), l'orientation du corps du robot (sous forme d'un quaternion) (x, y, z, w) et les positions angulaires de ses articulations. Sur 4 pas de temps futurs, on obtient 76 valeurs en tout, donc 76 neurones d'input.
- Le sous-réseau *policy* composé de 3 couches (512, 256 puis 12 neurones chacune). Ces couches sont reliées de manière *dense* avec des fonctions d'activation de type Rectified Linear Unit (ReLU). Le plus important à noter est que ce réseau donne en sortie les actions à effectuer : les 12 positions angulaires des articulations.
- Le sous-réseau *value function* de même architecture que le réseau *policy*. Ce sous réseau est propre aux algorithmes de type Actor-Critic. Comme vu en 3.4.1, plus spécifiquement dans l'équation 3.8, nous avons besoin d'un réseau de neurones pour estimer la valeur d'un couple (état, action à prendre dans cet état) pour le calcul de *l'avantage function*. Comme il prend en entrée les mêmes informations que le réseau *policy*, il forme ainsi un deuxième sous-réseau.

En ajoutant le nombre de neurones nécessaires pour ces deux sous-ensembles on obtient en tout 160 neurones d'entrée pour le réseau.

Pour rendre le réseau pilotable, nous avons pensé au transfer learning. L'idée est de reprendre le réseau ayant acquis la marche puis de l'entraîner une seconde fois en lui fournissant des entrées différentes. A la place des 4 futures positions du robot référence, je lui envoie une commande en vitesse (vitesse linéaire selon x, y et z et une vitesse angulaire roll, pitch, yaw (r,p,y) correspondant aux vitesses angulaires autour des axes x, y, z respectivement).

Je fais face à un problème, en me renseignant sur les prérequis pour faire du transfer learning et en discutant avec le docteur en IA de mon équipe, nous estimons que cette approche n'est pas la bonne. En effet, le transfer learning fonctionne si on respecte deux règles au préalable : la proximité entre la tâche acquise et celle que l'on veut acquérir avec le second entraînement et l'identicité entre le réseau de départ du second entraînement et le réseau final obtenu à la fin du premier.

Le premier point restreint la différence entre les compétences acquises du modèle pré-entraîné et les compétences souhaitées à acquérir lors du ré-entraînement. Par exemple, si j'ai un réseau entraîné qui sait marcher en ligne droite à vitesse constante, je ne peux pas espérer avoir de bons résultats si je demande au robot de faire un saut périlleux arrière. Il faut donc découper la progression du robot en plusieurs sous-tâches plus simples.

Le second point restreint l'architecture du réseau : on ne peut pas faire du transfer learning sur le modèle pré-entraîné auquel on enlève des entrées, ou certaines couches (cas particuliers).

Notre approche contredit donc ce second point : donner en entrée 6 commandes en vitesse à la place de 76 informations de la référence à suivre. La dimension des valeurs en entrée diminue (76 à 6) ainsi que leur nature (positions linéaires, angulaires, articulaires à vitesses linéaires). Pour continuer à utiliser le réseau tel quel (avec 76 informations décrivant la référence en entrée du réseau), il aurait fallu calculer, au préalable, via de la cinématique inverse, les positions des articulations en fonction de la vitesse voulue. Ce qui retirerai tout intérêt à notre réseau de neurones et à l'approche via l'IA. Avoir un réseau qui ne fait que de l'imitation n'est pas intéressant pour nous. Nous voulons un réseau capable de nous donner les positions des articulations en fonction de la vitesse du robot fournie en entrée sans autre intermédiaire.

Pour ceci, voici la nouvelle architecture que je vais utiliser :

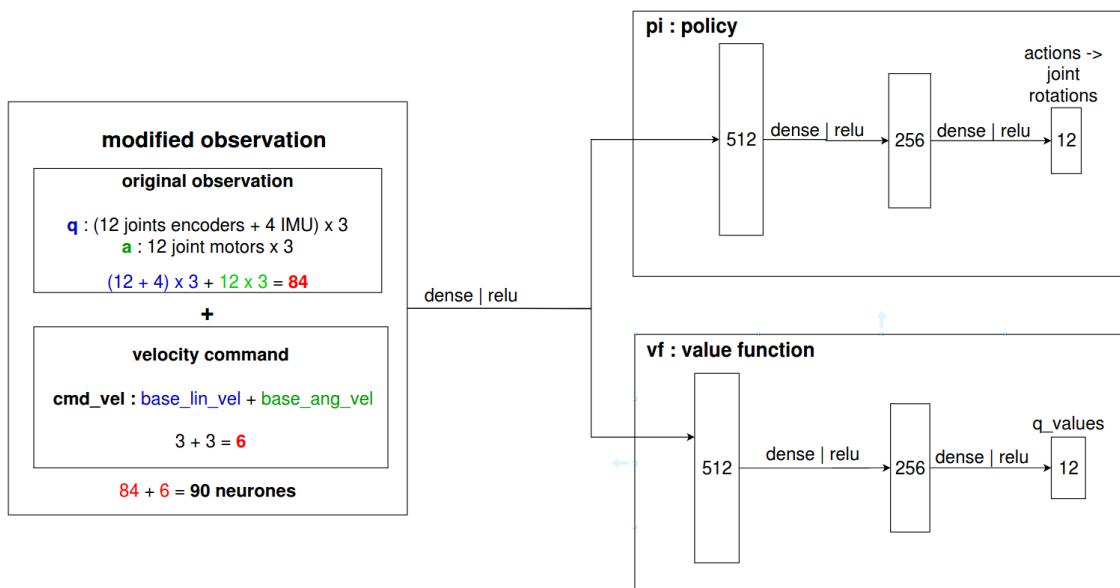


FIG. 4.5 : Architecture du réseau de neurones personnalisée : utilisant une commande en vitesse en entrée

L'algorithme d'entraînement reste le même : PPO, la reward function également. A l'issue de cette modification, on pourrait se dire que le réseau ne se sert plus de la référence pour apprendre à marcher et que des comportements non désirés pourraient faire surface. Ce n'est pas le cas, la référence reste utile et exploitée, mais lors de l'entraînement uniquement via la *reward function* qui reste toujours grossièrement une mesure de la différence entre le comportement du robot et de la référence du chien.

Aussi, afin de produire une commande en vitesse cohérente avec la référence utilisée dans la *reward function* (la référence marchant à 1.15 m.s^{-1} , donner une commande en vitesse à 2 m.s^{-1} serait contradictoire et n'aurait pas abouti à de bons résultats), j'ai du calculer les vitesses linéaires du robot à fournir en commande depuis les positions et orientations du corps du robot de référence. Ainsi, pour une référence marchant à $\approx 1.15 \text{ m.s}^{-1}$ je donnais une vitesse correspondante pour la commande du robot à entraîner.

Le changement d'entrée n'impacte presque pas les performances à l'issue de l'entraînement, et nous permet ainsi d'utiliser le réseau avec une commande classique en vitesse.

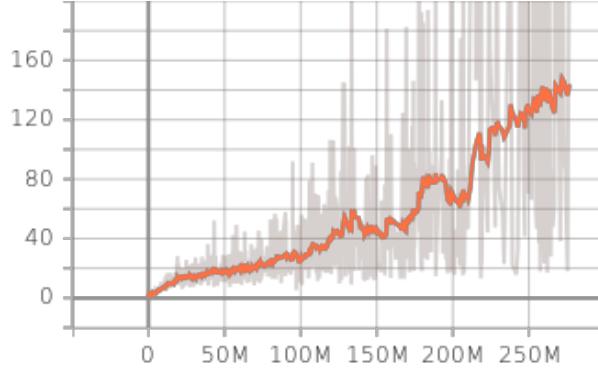


FIG. 4.6 : Architecture du réseau de neurones personnalisée : utilisant une commande en vitesse en entrée

Désormais nous avons à notre disposition un réseau pilotable en vitesse. Mais étant donné qu'il n'a été entraîné que sur un terrain plat et avec une trajectoire rectiligne uniforme, le robot ne peut pas tourner ni freiner, ni adapter sa marche pour enjamber un obstacle, etc.

4.3.3 La dérive du robot

D'autre part, lors du test du robot entraîné avec une commande en vitesse, je me suis rendu compte d'un phénomène de dérive du robot, plus il avance, plus il tend à s'éloigner de la référence et finit par tourner vers la droite (y positifs) de plus en plus *drift*.

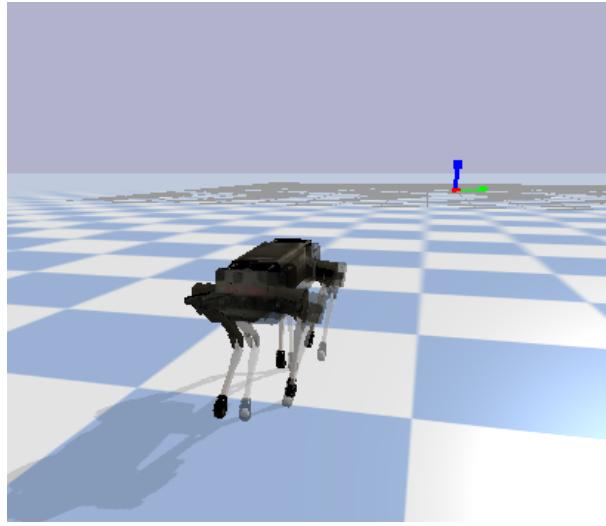


FIG. 4.7 : Dérive du robot sur sa droite (Repère «world» au loin comme référence)

Ce problème de dérive est aussi présent sur le code original. A première vue cette dérive est handicapante, car si le robot n'arrive pas à suivre avec précision une commande simple, il risque d'y avoir propagation de cette dérive lors de l'apprentissage de trajectoires plus complexes.

Notre réflexe a été de changer légèrement la fonction de reward, plus précisément les poids associés à chaque sous-récompense, pour donner plus d'importance à la sous-récompense récompensant la proximité entre l'orientation du corps de la référence et celle du robot. Voici les valeurs des poids :

$$w^p = 0.05, w^v = 0.05, w^e = 0.1, w^{rp} = 0.6, w^{rv} = 0.2 \quad (4.1)$$

Nous avons également décidé d'enlever la synchronisation de la référence : fonctionnalité qui force la référence à «recoller» en se téléportant, le corps du robot simulé toutes les secondes environ. Ceci a pour objectif d'augmenter le résultat de la différence entre positions, orientation, positions articulaires... entre référence et agent simulé étant donné l'absence de synchronisation.

Voici les résultats de cette expérience :

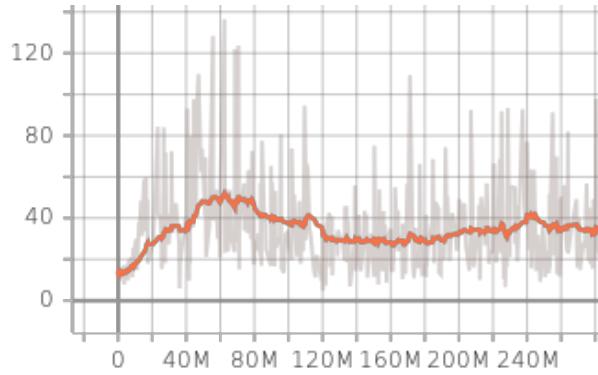


FIG. 4.8 : Courbe de reward pour un ré-entraînement avec fonction de reward modifiée et synchronisation de la référence désactivée

Nous constatons que l'agent ne s'améliore pas et voit son reward fortement et donc sa marche dégradée par rapport à une marche rectiligne classique (sans modifications du reward et avec synchronisation de la référence) (150 à 38) cf.4.6.

Nous avons ensuite formulé plusieurs idées pour réduire voire effacer cette dérive :

Une première remarque consiste à relever le fait qu'il n'y a pas de mesure de yaw en entrée du réseau. En effet, les 4 informations renvoyées par l'IMU sont : roll, pitch, roll rate et pitch rate. Au regard du calcul du reward, pour la sous-récompense en position et orientation, l'orientation du robot et de la référence sont directement lues à partir du simulateur et non pas à partir de l'IMU. Configurer l'IMU pour qu'elle renvoie (roll, pitch, yaw, roll rate, pitch rate, yaw rate) n'est donc pas utile pour le calcul du reward. Pour ce qui est de l'entrée du réseau, donner des informations supplémentaires peut se révéler contre productif. Ces informations agissant comme des distracteurs dans le cas où le yaw n'est pas utile à la marche. Un réseau de neurones est un approximateur de fonction, cherchant des corrélations entre entrées et sorties. Si une entrée n'est pas pertinente pour y corréler une action (sortie), il se peut que le réseau trouve une corrélation non causale et peut donc diminuer voire ruiner ses performances. Le deuxième argument contre l'utilité d'une observation incluant le yaw et yaw rate, pourrait que dans le cadre de l'apprentissage d'une marche rectiligne, le yaw est négligeable puisque la priorité est d'apprendre à tenir debout en équilibre, ce qui implique un contrôle sur le roll (chute sur le côté) et sur le

pitch (chute en avant, en arrière). Le yaw, en plus d'être déductible à partir du produit vectoriel $\vec{x} \wedge \vec{y}$ (**incorrect à vérifier**) paraît moins important voire inutile pour apprendre à maintenir un équilibre gauche droite - avant arrière. Dans le cas où on apprendrait une démarche de type pas de côtés, le yaw serait intéressant pour garder un cap pointant dans un sens différent que celui du sens de la marche. Cette hypothèse est vérifiable en lançant un entraînement avec le yaw et yaw rate ajouté à l'observation. Par soucis de temps et de disponibilité du serveur de calcul nous avons préféré reporter ce test pour nous concentrer sur une solution plus complète et intéressante.

Cette autre solution se base sur la manœuvrabilité du robot. L'hypothèse est la suivante : le robot ne peut pas corriger sa trajectoire pour recoller la référence s'il ne sait pas comment tourner en premier lieu. En effet, à la fin du premier entraînement, le robot n'a connu que la référence rectiligne. Pour imager, au cours de la «vie» du robot, il n'a appris qu'en se basant sur un modèle qui marchait droit. Son «savoir», quant à la marche se limite à la marche rectiligne : dans son «référentiel», il est «persuadé» de marcher droit. Alors que dans l'absolu, de notre point de vue, il dérive. Pour pallier ce problème il faut donc enrichir le «vécu» du robot avec des références variées, plus précisément des références qui effectuent des virages plus ou moins serrés à gauche et à droite. Pour cela il nous faut générer nos propres trajectoires de référence, n'ayant pas de chien à partir duquel extraire une motion capture. Les informations de la motion capture du chien sont lues et exploitées à partir d'un fichier, présentant un tableau. Chaque ligne représentant une image (frame) de mouvement du chien, et chaque colonne représentant un paramètre (position x y z, orientation x y z w, positions angulaires des articulations).

```
{
  "LoopMode": "Wrap",
  "FrameDuration": 0.01667,
  "EnableCycleOffsetPosition": true,
  "EnableCycleOffsetRotation": false,
  "Frames": [
    Erwin Coumans, a year ago · Initial source code accompanying the paper
  [
    [0.00000, 0.00000, 0.43701, 0.49491, 0.53393, 0.49912, 0.46997, -0.12721, 0.87675, -0.95545, -0.25301, 0.18682, -1.14403, -0.19362, 0.14638, -0.77823, -0.89528, 0.85437, -0.97596], ...
    [0.01641, 0.00223, 0.43771, 0.48959, 0.53669, 0.50119, 0.47918, 0.11820, -0.94606, -0.28172, 0.03357, -1.16456, -0.20247, 0.17747, -0.77184, -0.89744, -0.05174, -0.93399], ...
    [0.03278, 0.00476, 0.43896, 0.48274, 0.53845, 0.50530, 0.47984, -0.12518, 0.15584, -0.92492, -0.39683, -0.11684, -1.15057, -0.21314, 0.22216, -0.76688, -0.10566, -0.14981, -0.88721], ...
    [0.04882, 0.00706, 0.44055, 0.47656, 0.53895, 0.50939, 0.47217, -0.12177, 0.19977, -0.89664, -0.31713, 0.25529, -1.10780, -0.22449, 0.26899, 0.75917, -0.11668, 0.23940, -0.83319], ...
    [0.06588, 0.00883, 0.44216, 0.47995, 0.53858, 0.51304, 0.47428, -0.12887, 0.22336, -0.86296, -0.31289, -0.35342, -0.05187, -0.23665, 0.31297, -0.74463, -0.12989, -0.31595, -0.77429], ...
    [0.08286, 0.01033, 0.44397, 0.46634, 0.53774, 0.51543, 0.47717, -0.12319, 0.25093, -0.82652, -0.29625, -0.41374, -0.97984, -0.24941, 0.35253, -0.72424, -0.13472, -0.37088, -0.71971], ...
    [0.09884, 0.01171, 0.44605, 0.46371, 0.53747, 0.51525, 0.48922, -0.12780, 0.26864, -0.79123, -0.27896, -0.47452, -0.91527, -0.26153, 0.38053, 0.59634, -0.13817, 0.40471, -0.67348], ...
    [0.11564, 0.01337, 0.44783, 0.46172, 0.53682, 0.51397, 0.48422, -0.13582, 0.28660, -0.75117, -0.28068, -0.51479, -0.83989, -0.27053, 0.40688, 0.66892, -0.11715, 0.41160, -0.63590], ...
    [0.13247, 0.01557, 0.44866, 0.46017, 0.53653, 0.51145, 0.48867, -0.14270, 0.30399, -0.71655, -0.21667, -0.52983, -0.77176, -0.27539, 0.43360, -0.64668, -0.10623, 0.37913, -0.63582], ...
    [0.14967, 0.01760, 0.44988, 0.45870, 0.53617, 0.50991, 0.49298, -0.14760, 0.32386, -0.68749, -0.16653, -0.56854, -0.73515, -0.27867, 0.47223, -0.65266, -0.10817, -0.36617, -0.68954], ...
    [0.16688, 0.01936, 0.44679, 0.45801, 0.53697, 0.50846, 0.49332, -0.14926, 0.35277, -0.66859, -0.14587, -0.46223, -0.71830, -0.27985, 0.49227, -0.69369, -0.10386, -0.25300, -0.72486], ...
    [0.18457, 0.02113, 0.44491, 0.45884, 0.53861, 0.50979, 0.49131, -0.14788, 0.37881, -0.65994, -0.12952, -0.41299, -0.68721, -0.26795, 0.55542, -0.74544, -0.19247, -0.19779, -0.76888], ...
    [0.20315, 0.02255, 0.44338, 0.46159, 0.54101, 0.50733, 0.48666, -0.14168, 0.40261, -0.65184, -0.16246, -0.34246, -0.76643, -0.24415, 0.54899, 0.70846, -0.10484, 0.15274, -0.78151], ...
    [0.22228, 0.02351, 0.44195, 0.46521, 0.54355, 0.50982, 0.48898, -0.13629, 0.42854, -0.65674, -0.16512, 0.41743, -0.79699, -0.22199, 0.52110, -0.82139, -0.10887, -0.11558, -0.79425], ...
    [0.24349, 0.02428, 0.44024, 0.46907, 0.54564, 0.50727, 0.47426, -0.12753, 0.44960, -0.68466, -0.17788, -0.23585, 0.48482, -0.21058, 0.47836, -0.87137, -0.11339, -0.67894, -0.80477], ...
    [0.26441, 0.02473, 0.43823, 0.47325, 0.54703, 0.50712, 0.46868, -0.11968, 0.45798, -0.76842, -0.19387, -0.17536, 0.49564, -0.20651, 0.44393, 0.92752, -0.11522, -0.04088, -0.81465], ...
    [0.28558, 0.02464, 0.43626, 0.47835, 0.54689, 0.50616, 0.46464, -0.12197, 0.43880, -0.86342, -0.19421, -0.11435, 0.393678, -0.19062, 0.40079, -0.97485, -0.11837, -0.00483, -0.82884], ...
    [0.30563, 0.02398, 0.43551, 0.48404, 0.54571, 0.50948, 0.46196, -0.12253, 0.41137, -0.95532, -0.19119, -0.60919, -0.96439, -0.17122, 0.33514, -0.190492, -0.11824, 0.02713, -0.81663], ...
    [0.32565, 0.02348, 0.43543, 0.48863, 0.54314, 0.50928, 0.46192, -0.12639, 0.37667, -0.10692, -0.18348, -0.08853, -0.97989, -0.16188, 0.24489, 0.180477, -0.11752, 0.06483, -0.81530], ...
    [0.34513, 0.02264, 0.43600, 0.49248, 0.53972, 0.50166, 0.46315, -0.13757, 0.29075, -0.15527, -0.17385, 0.04504, -0.97993, -0.16889, 0.14852, -0.98629, -0.11866, 0.16292, -0.80885], ...
    [0.36352, 0.02156, 0.43678, 0.49651, 0.53508, 0.50804, 0.46553, -0.14822, 0.15752, -0.19552, -0.16643, 0.09081, -0.97221, -0.16121, 0.04991, 0.95571, -0.12166, 0.14042, -0.79969], ...
    [0.38133, 0.01988, 0.43339, 0.50053, 0.52944, 0.49854, 0.46971, -0.15680, 0.080492, -0.19387, -0.15419, 0.13039, -0.95309, -0.16926, -0.05029, -0.89983, -0.12653, 0.17847, -0.78722], ...
    [0.39841, 0.01837, 0.44059, 0.50384, 0.52281, 0.49739, 0.47479, -0.15983, -0.13783, -0.15844, -0.14030, 0.16753, -0.92657, -0.18165, 0.47682, -0.85218, -0.13280, 0.21770, -0.77474], ...
    [0.41513, 0.01669, 0.44294, 0.50649, 0.51629, 0.49694, 0.47954, -0.15788, -0.25191, -0.18942, 0.13183, 0.19619, -0.88943, -0.19374, -0.23443, -0.78796, -0.14329, 0.25547, -0.75380], ...
    [0.43177, 0.01473, 0.44521, 0.50987, 0.51037, 0.49701, 0.48396, -0.15548, -0.33574, -0.101839, -0.12408, 0.22932, -0.84860, -0.20385, -0.30441, -0.72873, 0.15418, 0.28667, -0.72651], ...
    [0.44692, 0.01314, 0.44766, 0.51097, 0.50622, 0.49819, 0.48428, -0.15480, -0.40217, -0.95462, -0.12135, 0.23766, -0.88623, -0.20927, -0.35212, -0.67100, -0.16646, 0.31242, -0.69448], ...
    [0.46489, 0.01140, 0.45821, 0.51169, 0.50451, 0.49981, 0.49356, -0.14767, -0.51588, -0.87697, -0.11954, 0.25998, -0.75784, -0.21183, -0.38106, -0.61525, -0.17028, 0.33364, -0.66814], ...
    [0.48202, 0.00922, 0.45259, 0.51313, 0.50273, 0.49885, 0.48834, -0.14749, -0.78665, -0.12894, 0.26968, -0.77915, -0.19579, -0.58446, -0.17731, 0.34955, -0.62396], ...
    [0.49978, 0.00862, 0.45301, 0.51399, 0.50995, 0.50649, 0.47889, 0.03442, -0.44226, -0.72261, -0.12344, 0.27518, -0.66283, -0.19484, -0.31802, -0.61017, -0.18553, 0.38462, -0.62481], ...
    [0.51793, 0.00852, 0.45156, 0.51416, 0.49704, 0.50999, 0.47885, -0.03327, -0.39805, -0.70482, -0.12444, 0.29981, -0.62361, -0.20894, -0.25119, 0.65274, -0.19997, 0.44064, -0.67343], ...
    [0.53748, 0.008395, 0.44913, 0.51411, 0.49529, 0.51164, 0.47813, -0.07110, -0.32111, -0.76216, -0.12382, 0.32680, -0.59567, -0.19979, -0.19735, -0.68530, -0.18261, 0.49555, -0.73822], ...
    [0.55659, 0.008280, 0.44697, 0.51485, 0.49644, 0.50985, 0.47885, -0.07549, -0.26493, -0.88317, -0.11877, 0.35467, -0.59078, -0.20209, -0.14852, -0.73239, 0.16125, 0.51232, -0.79071], ...
    [0.57577, 0.008167, 0.44587, 0.51650, 0.49783, 0.50699, 0.47789, -0.07131, -0.21348, -0.82662, -0.11386, 0.37332, -0.60466, -0.19963, -0.88971, -0.74582, -0.14228, 0.49488, -0.83879], ...
    [0.59497, 0.008154, 0.44312, 0.51800, 0.50013, 0.50333, 0.47774, -0.08576, -0.15562, -0.87133, -0.11535, 0.37462, -0.65372, -0.20273, -0.04893, -0.76238, 0.12839, 0.46938, -0.90473], ...
    [0.61400, 0.008191, 0.44160, 0.51843, 0.50324, 0.49974, 0.47774, -0.09484, -0.10177, -0.98486, -0.12731, 0.36711, -0.75677, -0.20671, -0.01317, -0.77285, -0.11790, 0.43685, -0.97024], ...
    [0.63303, 0.008233, 0.44009, 0.51746, 0.50557, 0.49713, 0.47985, -0.10425, -0.04639, -0.92873, -0.14730, 0.35852, -0.86271, -0.21267, 0.02886, -0.78466, -0.18501, 0.39879, -0.101703], ...
    [0.65155, 0.008263, 0.43886, 0.51586, 0.50831, 0.49492, 0.48016, -0.11146, 0.00547, -0.94503, -0.18348, 0.32598, -0.98252, -0.21663, 0.07145, -0.79413, -0.09216, 0.31946, -0.04176], ...
    [0.67085, 0.008126, 0.43824, 0.51299, 0.51174, 0.49342, 0.48113, -0.12947, 0.05387, -0.95210, -0.21892, 0.23998, -0.10764, -0.22485, 0.10828, -0.79239, -0.08403, 0.22582, -0.10413]
  ]
]
```

FIG. 4.9 : Contenu du fichier décrivant la marche d'un chien utilisée comme référence (38 frames et 19 paramètres)

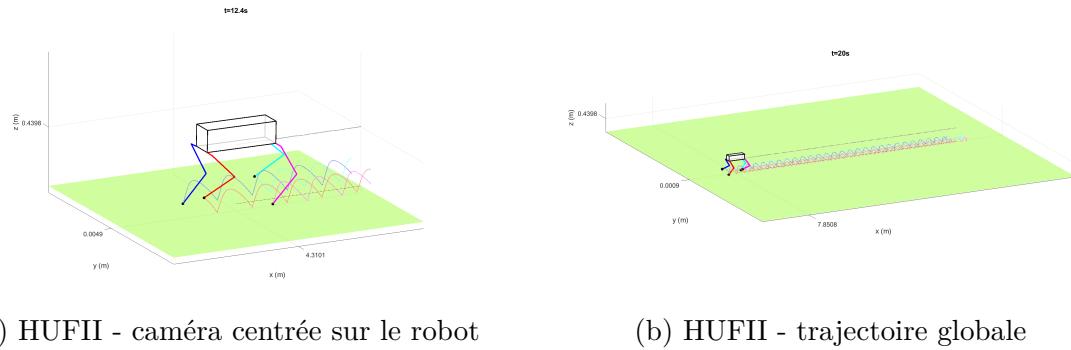


FIG. 4.10 : HUFII dans le simulateur Matlab Simulink

4.3.4 Manoeuvrabilité du robot

Grâce au travail du docteur en automatique et de son modèle de robot quadrupède sous Matlab Simulink, il est possible de créer notre propre robot de référence suivant une trajectoire personnalisée.

A partir de cette simulation, il est possible d'enregistrer toutes les informations nécessaires à la construction d'un fichier similaire. Cependant, la morphologie du robot quadrupède sous Matlab est différent de celui que je manipule sous Pybullet. En effet, le modèle du robot HUFII n'existant encore que sur un fichier de type CAO, il est difficile de le traduire à un format lisible et exploitable par Matlab ou Pybullet. Ainsi, le modèle du robot sous Simulink est une représentation simpliste, cela est aussi du au fait que Matlab ne propose pas d'outils de simulation aussi avancées que d'autres outils populaires en robotique : ROS Gazebo ou Pybullet par exemple.

Sous Matlab les dimensions sont donc similaires à celui du véritable robot HUFII, mais sous Pybullet le modèle de robot utilisé est le robot Laikago. Jusqu'ici, je n'avais pas besoin de coller parfaitement aux dimensions du robot HUFII, j'ai donc travaillé avec le robot Laikago en simulation. Maintenant, il faut donc accorder les deux simulations Matlab et Pybullet, afin que le fichier donné par la simulation Matlab soit cohérente avec le robot que j'utilise. Nous avons décidé de procéder ainsi car n'ayant pas de fichier au format urdf (format exploitable par Pybullet) du robot HUFII, il est plus simple de changer les dimensions du robot sur la simulation Matlab par les dimensions du robot Laikago. Cette étape, nous a pris plus de temps que prévu. Les deux simulateurs sont sensibles à des choses différentes, par exemple, sous Matlab les angles sont modulo 2π . Sur Pybullet, si on donne un angle supérieur à 2π , le moteur de l'articulation correspondante va essayer de faire un tour complet plus le surplus après 2π . Les mouvements du robot sont donc chaotiques et inexploitables. Nous avons lancé un entraînement avec cette référence doit voici les résultats :

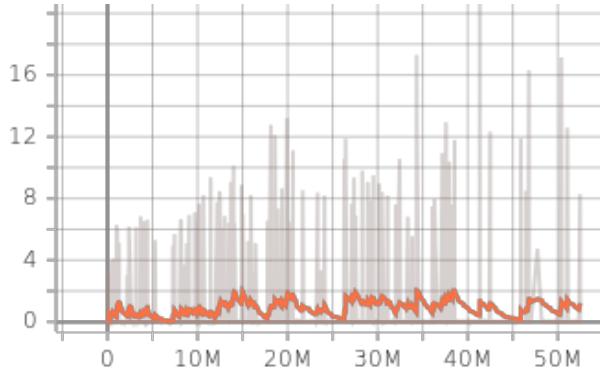


FIG. 4.11 : Evolution du reward avec une référence incorrecte : chutes brutales très fréquentes dès le début de la simulation

Une fois ces corrections faites, nous avons lancé un nouvel entraînement (sans réutiliser l'agent entraîné avec la motion capture) avec cette nouvelle référence personnalisée décrivant une trajectoire cyclique composée de :

- marche stationnaire pendant 2 secondes
- marche avant tout droit pendant 10 secondes

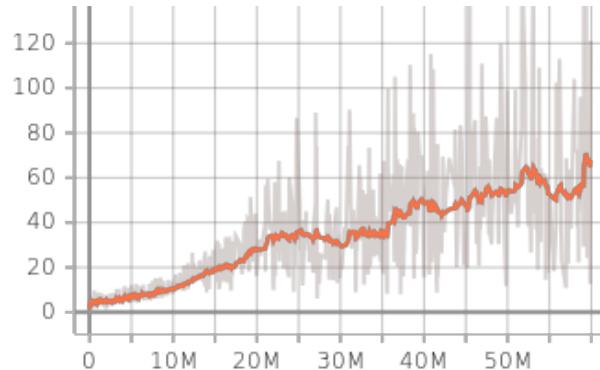


FIG. 4.12 : Evolution du reward avec une référence corrigée et personnalisée

Bien que le réseau soit encore en cours d'entraînement (60M timesteps sur 200M-250M) nous constatons que le robot, l'agent gagne des récompenses de manière croissante, et ce, à un rythme plus élevé que pour son homologue basé sur la motion capture. Ceci est notamment explicable par la nature plus stable, robotique de notre nouvelle référence par rapport à la motion capture du chien, moins stable, plus complexe et naturelle. Ce nouveau réseau sera utilisé comme base des prochains sur-entraînements, pour enrichir la palette de commandes acceptables par le robot, y compris la trajectoire en virage, potentiellement utile à la correction de la dérive.

4.3.5 Perspectives et travaux futurs

Pour compléter ce travail de marche quadrupède, il serait intéressant de passer à la version 3 de Stable Baselines, permettant un vrai réapprentissage, sans période de ré-adaptation de l'optimiseur. Cette version permettrait aussi de pouvoir passer sur du calcul

Chapitre 4. Travaux et réalisations

GPU et non plus CPU, plateforme de calcul beaucoup plus efficace pour l'entraînement de réseaux de neurones, grâce à ses bonnes propriétés de parallélisation.

Il restera aussi à développer la partie franchissement du robot : pentes, marches d'escaliers, terrain accidenté, instable, graviers, sable... Pouvant être traitées uniquement avec de la proprioception [3], ou alors nécessitant des capteurs plus avancés (Lidar, caméras) pour déterminer la nature du terrain, obstacles et décider de l'allure à suivre.

Bibliographie

- [3] J. LEE, J. HWANGBO, L. WELLHAUSEN, V. KOLTUN et M. HUTTER, « Learning Quadrupedal Locomotion over Challenging Terrain », *Science Robotics*, t. 5, n° 47, p. 1-22, 21 oct. 2020. DOI : [10.1126/scirobotics.abc5986](https://doi.org/10.1126/scirobotics.abc5986).
- [4] X. BIN PENG, E. COUMANS, T. ZHANG, T.-W. EDWARD LEE, J. TAN et S. LEVINE, « Learning Agile Robotic Locomotion Skills by Imitating Animals », rapp. tech., 2020. DOI : [10.15607/rss.2020.xvi.064](https://doi.org/10.15607/rss.2020.xvi.064).
- [5] J. HWANGBO, J. LEE, A. DOSOVITSKIY, D. BELLICOSO, V. TSOUNIS, V. KOLTUN et M. HUTTER, « Learning agile and dynamic motor skills for legged robots », *Science Robotics*, t. 4, n° 26, 16 jan. 2019. DOI : [10.1126/scirobotics.aau5872](https://doi.org/10.1126/scirobotics.aau5872).
- [6] V. TSOUNIS, M. ALGE, J. LEE, F. FARSHIDIAN et M. HUTTER, « DeepGait : Planning and Control of Quadrupedal Gaits Using Deep Reinforcement Learning », *IEEE Robotics and Automation Letters*, t. 5, n° 2, p. 3699-3706, 2020. DOI : [10.1109/LRA.2020.2979660](https://doi.org/10.1109/LRA.2020.2979660).
- [7] R. S. SUTTON et A. G. BARTO, *Reinforcement learning : An Introduction (2nd edition 2018)*, Bradford B, 9. 2018, t. 3.
- [8] V. MNIIH, K. KAVUKCUOGLU, D. SILVER, A. GRAVES, I. ANTONOGLOU, D. WIERSTRA et M. RIEDMILLER, « Playing Atari with Deep Reinforcement Learning », déc. 2013.
- [9] R. S. SUTTON, D. MCALLESTER, S. SINGH et Y. MANSOUR, « Policy Gradient Methods for Reinforcement Learning with Function Approximation », rapp. tech.
- [10] L. ANQIAO, W. ZHICHENG, W. JUN et Z. QIUGUO, « Bound Controller for a Quadruped Robot using Pre-Fitting Deep Reinforcement Learning », rapp. tech. 3, 2020.
- [11] T. HAARNOJA, S. HA, A. ZHOU, J. TAN, G. TUCKER et S. LEVINE, « Learning to Walk via Deep Reinforcement Learning », rapp. tech., 0.
- [12] Y. YANG, K. CALUWAERTS, A. ISCEN, T. ZHANG, J. TAN et V. SINDHWANI, « Data efficient reinforcement learning for legged robots », *arXiv*, 2019.
- [13] J. SCHULMAN, F. WOLSKI, P. DHARIWAL, A. RADFORD et O. KLIMOV, « Proximal policy optimization algorithms », *arXiv*, 2017.
- [14] J. SCHULMAN, S. LEVINE, P. MORITZ, M. JORDAN et P. ABBEEL, « Trust region policy optimization », t. 3, 2015, p. 1889-1897.
- [16] V. MNIIH, A. P. BADIA, L. MIRZA, A. GRAVES, T. HARLEY, T. P. LILlicrap, D. SILVER et K. KAVUKCUOGLU, « Asynchronous methods for deep reinforcement learning », t. 4, 2016, p. 2850-2869.

Bibliographie

- [18] J. TAN, T. ZHANG, E. COUMANS, A. ISCEN, Y. BAI, D. HAFNER, S. BOHEZ et V. VANHOUCKE, « Sim-to-Real : Learning Agile Locomotion For Quadruped Robots », *arXiv*, 26 avr. 2018.
- [19] W. YU, C. K. LIU et G. TURK, « Policy Transfer with Strategy Optimization », *arXiv*, 12 oct. 2018.
- [20] N. HANSEN, A. OSTERMEIER et A. GAWELCZYK, « On the adaptation of arbitrary normal mutation distributions in evolution strategies : The generating set adaptation », *Proceedings of the Sixth International Conference on Genetic Algorithms*, p. 57-64, 1995.
- [22] G. BROCKMAN, V. CHEUNG, L. PETTERSSON, J. SCHNEIDER, J. SCHULMAN, J. TANG et W. ZAREMBA, « Openai gym », *arXiv preprint arXiv :1606.01540*, 2016.
- [23] E. COUMANS et Y. BAI, « Pybullet, a python module for physics simulation for games, robotics and machine learning », 2016.

Webographie

- [1] *Capgemini Engineering Webpage.* adresse : <https://www.capgemini.com/fr-fr/service/capgemini-engineering/>.
- [2] MULTI-AUTEURS, *Wikipedia - Capgemini.* adresse : <https://fr.wikipedia.org/wiki/Capgemini>.
- [15] T. SIMONINI, *An introduction to Policy Gradients with Cartpole and Doom*, 2018. adresse : <https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f/>.
- [17] ——, *An intro to Advantage Actor Critic methods : let's play Sonic the Hedgehog !*, [Online ; accessed 2021-04-30], 26 juil. 2018. adresse : <https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/>.
- [21] A. HILL, A. RAFFIN, M. ERNESTUS, A. GLEAVE, A. KANERVISTO, R. TRAORE, P. DHARIWAL, C. HESSE, O. KLIMOV, A. NICHOL, M. PLAPPERT, A. RADFORD, J. SCHULMAN, S. SIDOR et Y. WU, *Stable Baselines*, <https://github.com/hill-a/stable-baselines>, 2018.

Annexes