gz-unitree: Reinforcement learning en robotique avec validation par moteurs de physique multiples pour le H1v2 d'Unitree

Gwenn Le Bihan
gwenn.lebihan@etu.inp-n7.fr
ENSEEIHT

10 Novembre 2025

1 Remerciements

Table des matières

1 Remerciements 2
2 Contexte 3
2.1 Bases théoriques du *Reinforcement Learning* 3
2.1.1 L'entraînement 3
2.1.1.1 Deep Reinforcement Learning 5
2.1.2 Tendances à la « tricherie » des agents 5
2.1.2.1 Sous-spécification de la fonction coût 5
2.1.2.2 Bug dans un moteur de physique 5
2.1.2.3 La validation comme méthode de mitigation 6
2.2 Application en robotique 6
2.3 Le H1v2 d'*Unitree* 6
2.4 Environnements et moteurs de simulation physique 6
2.4.1 MuJoCo 6
2.4.2 Gazebo 6
2.5 Reproductibilité logicielle 6
3 Packaging reproductible avec Nix 6
3.1 Reproductibilité 6
3.1.1 État dans le domaine de la programmation 6
3.1.2 Contenir les effets de bords 7
3.1.3 État dans le domaine de la robotique 7
3.1.4 Environnements de développement 7
3.2 Nix, le gestionnaire de paquets pur 8
3.2.1 Un *DSL*[1] fonctionnel 8
3.2.2 Un ecosystème de dépendances 9
3.2.3 Une compilation dans un environnement fixé 9
3.2.3.1 Un complément utile: compiler en CI 9
3.3 NixOS, un système d'exploitation à configuration déclarative 9
4 Étude du SDK d'Unitree et du bridge SDK ⇆ MuJoCo 11
4.1 Une base de code partiellement open-source 11
4.2 Canaux DDS bas niveau 11
4.3 Rétroingénierie des binaires 11
4.4 Un autre bridge existant: `unitree_mujoco` 11
5 Développement du bridge SDK ⇆ Gazebo 11
5.1 Établissement du contact 11
5.2 Réception des commandes 11
5.3 Émission de l'état 11
5.4 Essai sur des politiques réelles 11
5.5 Amélioration des performances 11
5.6 Enregistrement de vidéos 11
5.6.1 Contrôle programmatique de l'enregistrement 11

[1] Domain-Specific Language

5.7 Mise en CI/CD 11
5.7.1 Une image de base avec Docker 11
5.7.2 Une pipeline Github Actions 11
Bibliographie 11
Annexes 13

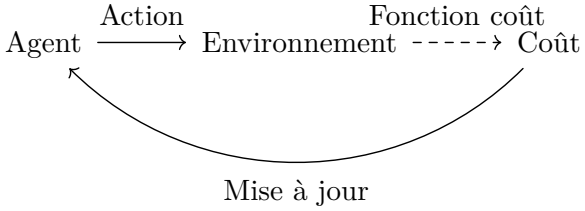
2 Contexte

2.1 Bases théoriques du *Reinforcement Learning*

L'apprentissage par renforcement, ou *Reinforcement Learning*, permet de développer des programmes sans expliciter leur logique: on décrit plutôt quatre choses, qui vont permettre à la logique d'émerger pendant la phase d'entraînement:

- Un *agent*: c'est le programme que l'on souhaite créer
- Des *actions* que l'agent peut choisir d'effectuer ou pas
- Un *environnement*, que les actions viennent modifier
- Un *coût* (ou *récompense*) qui dépend de l'environnement

La phase d'apprentissage consiste à trouver, par des cycles d'essai/erreur, quelles sont les meilleures actions à prendre en fonction de l'environnement actuel, avec meilleur définit comme « qui minimise le coût » (ou maximise la récompense):



Cette technique est particulièrement adaptée au problèmes qui se prêtent à une modélisation type « jeu vidéo », dans le sens où l'agent représente le personnage-joueur, et le coût un certain score, qui est condition de victoire ou défaite.

En robotique, on a des correspondances claires pour ces quatres notions:

Agent	Robot pour lequel on développe le programme de contrôle (appelée une *politique*)
Actions	Envoi d'ordres aux moteurs
Environnement	Le monde réel. C'est de loin la partie la plus difficile à simuler informatiquement. On utilise des moteurs de simulation physique, dont la multiplicité des implémentations est importante, voir Chapitre 2.4
Coût	un ensemble de contraintes (« ne pas endommager le robot »), dont la plupart dépendent de l'objectif de la politique

2.1.1 L'entraînement

Une fois que ce cadre est posé, il reste à savoir *comment* l'on va trouver la fonction qui associe un état de l'environnement à une action.

Une première approche naïve, mais suffisante dans certains cas, consiste à faire une recherche exhaustive et à stocker dans un simple tableau la meilleure action à faire en fonction d'un état de l'environnement:

État actuel (x, retour)	**Meilleure action** $+1$ ou -1	**Coûts associés**
$(0, \text{C'est plus})$		
$(1, \text{C'est plus})$		
$(3, \text{C'est moins})$		
$(4, \text{C'est moins})$		
$(5, \text{C'est moins})$		

Tableau 1. – Exemple d'agent à mémoire exhaustive pour un « C'est plus ou c'est moins » dans $\{0, 1, 2\}$, avec pour solution 2

L'entraînement consiste donc ici en l'exploration de l'entièreté des états possibles de l'environnement, et, pour chaque état, le calcul du coût associé à chaque action possible.

Il faut définir la fonction de coût, souvent appelée L pour *loss*:

$$L : E \to S \tag{1}$$

avec E l'ensemble des états possibles de l'environnement, et S un ensemble muni d'un ordre total (on utilise souvent $[0, 1]$)

Quand on parle de « coût d'une action », on parle du coût de l'état résultant de l'application de l'action en question à l'état actuel

On remplit la colonne « Action à effectuer » avec l'action au coût le plus bas:

État actuel (x, retour)	**Meilleure action** $+1$ ou -1	**Coûts associés** avec $L = (x, \text{retour}) \mapsto \lvert x - 2 \rvert$
$(0, \text{C'est plus})$	$+1$	$L(x+1,) = 2 \quad L(x-1,) = 2$
$(1, \text{C'est plus})$	$+1$	$L(x+1,) = 1 \quad L(x-1,) = 2$
$(3, \text{C'est moins})$	-1	$L(x+1,) = 2 \quad L(x-1,) = 3$
$(4, \text{C'est moins})$	-1	$L(x+1,) = 3 \quad L(x-1,) = 4$
$(5, \text{C'est moins})$	-1	$L(x+1,) = 4 \quad L(x-1,) = 5$

Tableau 2. – Entraînement terminé, avec pour fonction coût L la distance à la solution

Ici, cette approche exhaustive suffit parce que l'ensemble des états possibles de l'environnement, E, posssède 6 éléments

Cependant, ces ensembles sont bien souvent prohibitivement grands (e.g. $x \in [\![0, 10^{34}]\!]$), infinis ($x \in \mathbb{N}$) ou indénombrables ($x \in \mathbb{R}$)

Dans le cas de la robotique, E est une certaine représentation numérique du monde réel autour du robot, on imagine donc bien qu'il y a beaucoup trop d'états possibles.

Deep Reinforcement Learning

Une façon de remédier à ce problème de dimensions est de remplacer le tableau exhaustif par un réseau de neurones:

État actuel	devient la couche d'entrée
Meilleure action	devient la couche de sortie
Coûts associés	deviennent les neurones des couches cachées
Le remplissage du tableau	devient la rétropropagation pendant l'entraînement

2.1.2 Tendances à la « tricherie » des agents

Expérimentalement, on sait que des tendances « tricheuses » émergent facilement pendant l'entraînement [Réf. nécéssaire]: l'agent découvre des séries d'actions qui causent un bug avantageux vis à vis du coût associé, soit parce qu'il y a un bug dans le calcul de l'état de l'environnement post-action, soit parce que la fonction coût ne prend pas suffisemment bien en compte toutes les possibilités de l'environnement (autrement dit, il manque de contraintes).

Sous-spécification de la fonction coût

(Note: Bof cette partie)

Un exemple populaire est l'expérience de pensée du Maximiseur de trombones [1]: un agent avec pour environnement le monde réel, pour actions « prendre des décisions »; « envoyer des emails »; etc. et pour fonction récompense (une fonction à maximiser au lieu de minimiser) « le nombre de trombones existant sur Terre », finirait possiblement par réduire en escalavage tout être vivant capable de produire des trombones: la fonction coût est sous-spécifiée

Bug dans un moteur de physique

Dans le contexte de la robotique, le calcul de l'état post-action de l'environnement est le travail du *moteur de physique*.

Bien évidemment, ce sont des programmes complexes avec souvent des résolutions numériques d'équation physiques, il est presque inévitable que des bugs se glissent dans ces programmes.

Ces phénomènes, appelés *« glitches »* dans le jargon du jeu vidéo, peuvent se manifester de diverses manières:

(Note: Compliqué sans vidéo… ptet à remplacer par une phrase seulement, ou alors c'est peut-être déjà assez clair sans exemples?)

- Le passage à travers un objet solide à cause de cas limites dans les calculs de collision joueur-objet (appelé *No clip*)
- La téléportation du joueur sur des grandes distances sans cause raisonnable, souvent causé par des erreurs dans le calcul des coordonnées de sa position

- La projection d'un objet a une vitesse extrême, souvent causé par des cas limites dans le calcul de la vélocité lors d'une collision

Bien évidemment, pour l'agent, tant qu'un bug n'est pas explicitement découragé par sa prise en compte dans la fonction coût, si l'état résultant améliore le score, l'agent apprendra à faire cette action quand c'est utile.

(Note: Rien à voir mais je me dis, c'est enfait un moyen de trouver des bugs dans un physics engine ! ça me fait penser au Fuzzing un peu, mais avec un NN plutôt que du hasard contrôlé)

La validation comme méthode de mitigation (Note: ça se dit mitigation en français?)

2.2 Application en robotique

2.3 Le H1v2 d'*Unitree*

2.4 Environnements et moteurs de simulation physique

2.4.1 MuJoCo

2.4.2 Gazebo

2.5 Reproductibilité logicielle

3 Packaging reproductible avec Nix

3.1 Reproductibilité

3.1.1 État dans le domaine de la programmation

La différence entre une fonction au sens mathématique et une fonction au sens programmatique consiste en le fait que, par des raisons de practicité, on permet aux `functions` des langages de programmation d'avoir des *effets de bords*. Ces effets affectent, modifient ou font dépendre la fonction d'un environnement global qui n'est pas explicitement déclaré comme une entrée (un argument) de la fonction en question [2].

Cette liberté permet, par exemple, d'avoir accès à la date et à l'heure courante, interagir avec un système de fichier d'un ordinateur, générer une surface pseudo aléatoire par bruit de Perlin, etc.

Mais, en contrepartie, on perd une équation qui est fondamentale en mathématiques:

$$\forall E, F, \forall f : E \to F, \forall (e_1, e_2) \in E^2, e_1 = e_2 \Rightarrow f(e_1) = f(e_2) \tag{2}$$

En programmation, on peut très facilement construire un f qui ne vérifie pas ceci:

```
from datetime import date

def f(a):
  return date.today().year + a
```

Selon l'année dans laquelle nous sommes, `f`(0) n'a pas la même valeur.

De manière donc très concrète, si cette fonction `f` fait partie du protocole expérimental d'une expérience, cette expérience n'est plus reproductible, et ses résultats sont donc potentiellement

non vérifiables, si le papier est soumis le 15 décembre 2025 et la *peer review* effectuée le 2 janvier 2026.

3.1.2 Contenir les effets de bords

En dehors du besoin de vérifiabilité du monde de la recherche, la reproductibilité est une qualité recherchée dans certains domaines de programmation [3]

Il existe donc depuis longtemps des langages de programmation dits *fonctionnels*, qui, de manière plus ou moins stricte, limite les effets de bords. Certains langages font également la distinction entre une fonction *pure*[2] et une fonction classique [4]. Certaines fonctions, plutôt appelées *procédures*, sont uniquement composées d'effet de bord puisqu'elle ne renvoie pas de valeur [5]

3.1.3 État dans le domaine de la robotique

En robotique, pour donner des ordres au matériel, on intéragit beaucoup avec le monde extérieur (ordres et lecture d'état de servo-moteurs, flux vidéo d'une caméra, etc), souvent dans un langage plutôt bas-niveau, pour des questions de performance et de proximité abstractionnelle au matériel

De fait, les langages employés sont communément C, C++ ou Python[3] [6], des langages bien plus impératifs que fonctionnels [7].

L'idée de s'affranchir d'effets de bords pour rendre les programmes dans la recherche en robotique reproductibles est donc plus utopique que réaliste.

3.1.4 Environnements de développement

Cependant, ce qui fait un programme n'est pas seulement son code: surtout dans des langages plus anciens sans gestion de dépendance simple, les dépendances (bibliothèques) du programme, ainsi que l'environnement et les étapes de compilation de ce dernier, représentent également une partie considérable de la complexité du programme (par exemple, en C++, on utilise un outil générant des fichiers de configuration pour un autre outil qui à son tour configure le compilateur de C++ [8])

C'est cette partie que Nix, le gestionnaire de paquet, permet d'encapsuler et de rendre reproductible. Dans ce modèle, la compilation (et de manière plus générale la construction, ou *build*) du projet est la fonction que l'on veut rendre pure. L'entrée est le code source, et le résultat de la fonction est un binaire, qui ne doit dépendre que du code source.

$$\forall\ \mathrm{src}, \mathrm{bin}, \forall f \in \mathrm{bin}^{\mathrm{src}}, \forall (P_1, P_2) \in \mathrm{src}^2, P_1 = P_2 \Rightarrow f(P_1) = f(P_2) \tag{3}$$

Ici, P_1 et P_2 sont deux itérations du code source (src) du programme. Si le code source est identique, les binaires résultants de la compilation (f) sont égaux, au sens de l'égalité bit à bit.

On a la proposition (1), avec $E = \mathrm{src}$, l'ensemble des code source possibles pour un langage, et $F = \mathrm{bin}$, l'ensemble des binaires éxécutables

Nix ne peut pas garantir que le programme sera sans effets de bords au *runtime*, mais vise à le garantir au *build-time*.

[2] sans effets de bord

[3] Il arrive assez communément d'utiliser Python, un langage haut-niveau, mais c'est dans ce cas à but de prototypage, et le code contrôlant les moteurs est écrit dans un langage bas niveau plus appelé par Python par FFI

3.2 Nix, le gestionnaire de paquets pur

3.2.1 Un *DSL*[4] fonctionnel

Une autre caractéristique que l'on trouve souvent dans la famille de langages fonctionnels est l'omniprésence des *expressions*: quasi toute les constructions syntaxiques forment des expressions valides, et peuvent donc servir de valeur

`def g(x, y):`
`  if y == 5:`
`  x = 6`
`  else:`
`  x = 8`
`  return f(x)`
	`let g x y = f (`
	`  if y = 5 then`
	`  6`
	`  else`
	`  8`
	`)`

Python (`if` et `else` sont des instructions)	**OCaml** (`if` et `else` forment une expression)

Afin de décrire les dépendances d'un programme, l'environnement de compilation, et les étapes pour le compiler (en somme, afin de définir le $f \in \text{bin}^{\text{src}}$), Nix comprend un langage d'expressions [9]. Un fichier `.nix` définit une fonction, que Nix sait exécuter pour compiler le code source.

Expression d'une fonction en Python	En Nix
`lambda f(a): a + 3`	`{ a }: a + 3`

Voici un exemple de définition d'un programme, appelée *dérivation* dans le jargon de Nix:

```
{
  src-odri-masterboard-sdk,

  lib,
  stdenv,
  jrl-cmakemodules,
  cmake,
  python3Packages,
  catch2_3,
}:

stdenv.mkDerivation {
  pname = "odri_master_board_sdk";
  version = "1.0.7";

  src = src-odri-masterboard-sdk;

  preConfigure = ''
    cd sdk/master_board_sdk
  '';

  doCheck = true;

  cmakeFlags = [
    (lib.cmakeBool "BUILD_PYTHON_INTERFACE" stdenv.hostPlatform.isLinux)
  ];

  nativeBuildInputs = [
    jrl-cmakemodules
    python3Packages.python
    cmake
  ];
```

[4] Domain-Specific Language

```
  buildInputs = with python3Packages; [ numpy ];

  nativeCheckInputs = [ catch2_3 ];

  propagatedBuildInputs = with python3Packages; [ boost ];
}
```

La dérivation ici prend en entrée le code source (`src-odri-masterboard-sdk`), ainsi que des dépendances, que ce soit des fonctions relatives à Nix même (comme `stdenv.mkDerivation`) pour simplifier la définition de dérivation, ou des dépendances au programmes, que ce soit pour sa compilation ou pour son exécution (dans ce dernier cas de figures, les dépendances sont inclues ou reliées au binaire final)

3.2.2 Un ecosystème de dépendances

Afin de conserver la reproductibilité même lorsque l'on dépend de libraries tierces, ces dépendances doivent également avoir une compilation reproductible: on déclare donc des dépendances à des *packages* Nix, disponibles sur *Nixpkgs* [10].

Parfois donc, écrire un paquet Nix pour son logiciel demande aussi d'écrire les paquets Nix pour les dépendances de notre projet, si celles-ci n'existent pas encore, et cela récursivement. On peut ensuite soumettre nos paquets afin que d'autres puissent en dépendre sans les réécrire, en contribuant à *Nixpkgs* [11]

Pour ne pas avoir à compiler toutes les dépendances soit-même quand on dépend de `.nix` de *nixpkgs*, il existe un serveur de cache, qui propose des binaires des dépendances, Cachix [12]

3.2.3 Une compilation dans un environnement fixé

Certains aspects de l'environnement dans lequel l'on compile un programme peuvent faire varier le résultat final. Pour éviter cela, Nix limite au maximum les variations d'environnement. Par exemple, la date du système est fixée au 0 UNIX (1er janvier 1990): le programme compilé ne peut pas dépendre de la date à laquelle il a été compilé.

Quand le *sandboxing* est activé, Nix isole également le code source de tout accès au réseau, aux autres fichiers du système (ainsi que d'autres mesures) pour améliorer la reproductibilité [13]

Un complément utile: compiler en CI

Pour aller plus loin, on peut lancer la compilation du paquet Nix en *CI*[6], c'est-à-dire sur un serveur distant au lieu de sur sa propre machine. On s'assure donc que l'état de notre machine de développement personnelle n'influe pas sur la compilation, puisque chaque compilation est lancée dans une machine virtuelle vierge [14].

3.3 NixOS, un système d'exploitation à configuration déclarative

Une fois le programme compilé avec ses dépendances, il est prêt à être transféré sur l'ordinateur ou la carte de contrôle embarquée au robot.

Lorsqu'il y a un ordinateur embarqué, comme par exemple une Raspberry Pi [15], il faut choisir un OS sur lequel faire tourner le programme.

La encore, un OS s'accompagne d'un amas considérable de configuration des différentes parties du système: accès au réseau, drivers,…

[6]Continuous Integration, lit. intégration continue

Sur les OS Linux classiques tels que Ubuntu ou Debian, cette configuration est parfois stockée dans des fichiers, ou parfois retenue en mémoire, modifiée par l'execution de commandes.

C'est un problème assez récurrent dans Linux de manière générale: d'un coup, le son ne marche plus, on passe ½h sur un forum à copier-coller des commandes dans un terminal, et le problème est réglé… jusqu'à ce qu'il survienne à nouveau après un redémarrage ou une réinstallation.

Ici, NixOS assure que toute modification de la configuration d'un système est *déclarée* (d'où l'adjectif « déclaratif ») dans des fichiers de configurations, également écrit dans des fichiers `.nix` [16].

Ici encore, cela apporte un gain en terme de reproductibilité: l'état de configuration de l'OS sur lequel est déployé le programme du robot est, lui aussi, rendu reproductible.

4 Étude du SDK d'Unitree et du bridge SDK ⇆ MuJoCo

4.1 Une base de code partiellement open-source

4.2 Canaux DDS bas niveau

4.3 Rétroingénierie des binaires

4.4 Un autre bridge existant: `unitree_mujoco`

5 Développement du bridge SDK ⇆ Gazebo

5.1 Établissement du contact

5.2 Réception des commandes

5.3 Émission de l'état

5.4 Essai sur des politiques réelles

5.5 Amélioration des performances

5.6 Enregistrement de vidéos

5.6.1 Contrôle programmatique de l'enregistrement

5.7 Mise en CI/CD

5.7.1 Une image de base avec Docker

5.7.2 Une pipeline Github Actions

Bibliographie

[1] Nick Bostrom, « Ethical Issues in Advanced Artificial Intelligence », 2003, *Int. Institute of Advanced Studies in Systems Research and Cybernetics*. Consulté le: 8 octobre 2025. [En ligne]. Disponible sur: https://nickbostrom.com/ethics/ai

[2] Brian Lonsdorf, « Professor Frisby's Mostly Adequate Guide to Functional Programming », 2015, *Github*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://github.com/MostlyAdequate/mostly-adequate-guide/blob/master/ch03.md

[3] « Reproducible Builds ». Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://reproducible-builds.org/

[4] Fortran 2015 Committee Draft (J3/17-007r2), *ISO/IEC JTC 1/SC 22/WG5/N2137*. International Organization for Standardisation, 2017, p. 336-338. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://wg5-fortran.org/N2101-N2150/N2137.pdf

[5] « Relationship Between Routines, Functions, and Procedures », 13 janvier 2025, *IBM*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.ibm.com/docs/en/informix-servers/15.0.0?topic=statement-relationship-between-routines-functions-procedures

[6] « Different Types of Robot Programming Languages », 2015, *Plant Automation Technology*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.plantautomation-technology.com/articles/different-types-of-robot-programming-languages

[7] « Imperative programming: Overview of the oldest programming paradigm », 21 mai 2021, *IONOS*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/

[8] Bill Hoffman et Kenneth Martin, *The Architecture of Open Source Applications (Volume 1) CMake*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://aosabook.org/en/v1/cmake.html

[9] Consulté le: 19 mai 2025. [En ligne]. Disponible sur: https://nix.dev/manual/nix/2.17/language/

[10] Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://search.nixos.org/packages

[11] NixOS Wiki Authors, « Nixpkgs/Contributing ». Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://wiki.nixos.org/wiki/Nixpkgs/Contributing

[12] « Cachix — Nix binary cache hosting ». Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://www.cachix.org/

[13] « Nix (package manager) — Sandboxing ». Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://wiki.nixos.org/wiki/Nix_(package_manager)#Internals

[14] « GitHub-hosted runners », *Github*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://docs.github.com/en/actions/concepts/runners/github-hosted-runners

[15] Consulté le: 6 juin 2025. [En ligne]. Disponible sur: https://www.raspberrypi.com/

[16] Fernando Borretti, « NixOS for the Impatient », 7 mai 2023. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://borretti.me/article/nixos-for-the-impatient

Annexes