Apprentissage par renforcement de la locomotion d'un hexapode

Stage de Master 1, ENS Paris-Saclay

Alexandre IOOSS

Juin à Août 2020

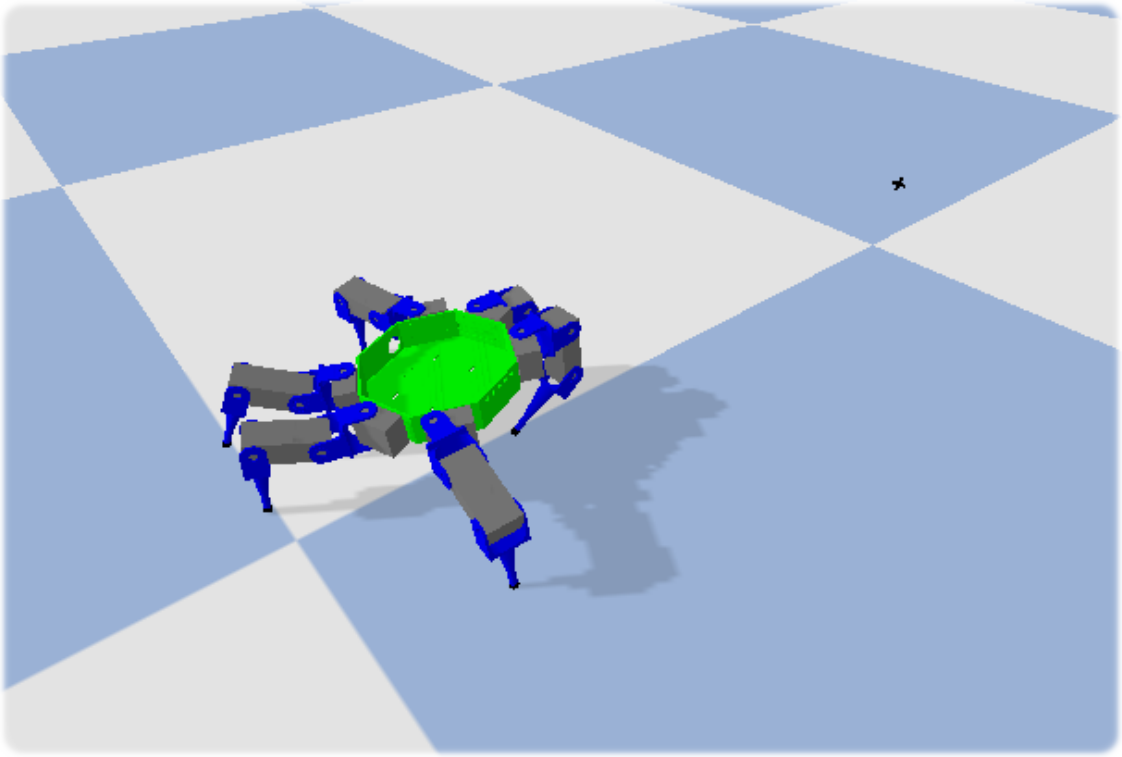

Table des matières

1 Présentation du sujet et problématique

Réaliser le contrôle de la locomotion d'un robot peut être une tâche difficile selon la structure du robot. En pratique ce problème est souvent résoluble en trouvant une commande optimale. Néanmoins, dans le cadre des récentes avancées en apprentissage machine par renforcement, il est intéressant d'explorer **l'apprentissage de la locomotion entièrement par renforcement**.

Le principe de l'apprentissage par renforcement est d'améliorer au fur et à mesure une politique de contrôle. Ainsi un agent va interagir avec le robot et suivre cette politique tout en collectant des données nécessaires pour améliorer ses performances, ici en maximisant une fonction de gain.

Lors d'un travail de recherche au cours de l'année de Master 1, nous avons pu construire un robot hexapode dans le but de réaliser sa simulation. À la fin de ce projet nous avons obtenu un robot fonctionnel et un prototype de simulation physique tout en ayant exploré les technologies et algorithmes à l'état de l'art de l'apprentissage par renforcement profond. Nous vous recommandons de lire le rapport de ce travail encadré de recherche disponible au lien suivant : https://kraby.readthedocs.io/en/latest/Rapport_de_TER.pdf

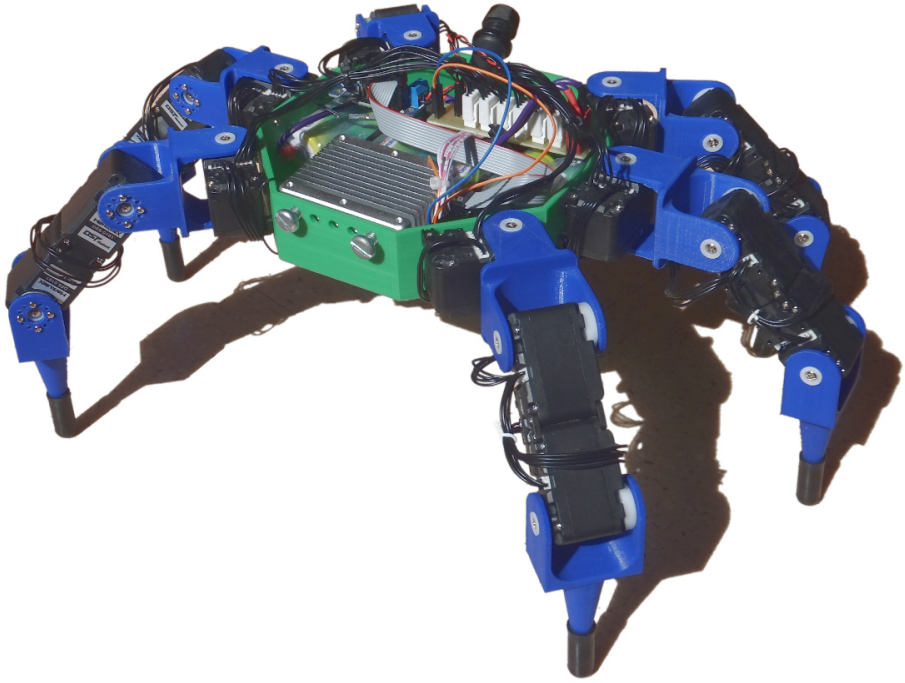

FIG. 1 : Robot hexapode sur lequel on va travailler

Dans le contexte de la crise sanitaire, nous avons choisi de continuer à avancer sur ce projet en télétravail tout en étant encadré par Anthony Juton (ENS Paris-Saclay) et Justin Carpentier (INRIA Paris, groupe WILLOW).

Le but principal a été de **mettre en place l'apprentissage sur une patte du robot**. Cela a alors conduit à :

- mettre en place des logiciels embarqués sur le robot hexapode pour pouvoir contrôler ses moteurs et récupérer les données des différents capteurs ;
- finir la simulation physique du robot hexapode ;
- mettre en place un algorithme d'apprentissage profond et étudier l'effet des différents hyperparamètres lors de l'apprentissage ;
- prendre en main une ferme de calculs et les outils associés.

Le suivi de ce stage a été effectué en organisant régulièrement des échanges par courriels et visioconférences afin d'échanger sur le travail effectué.

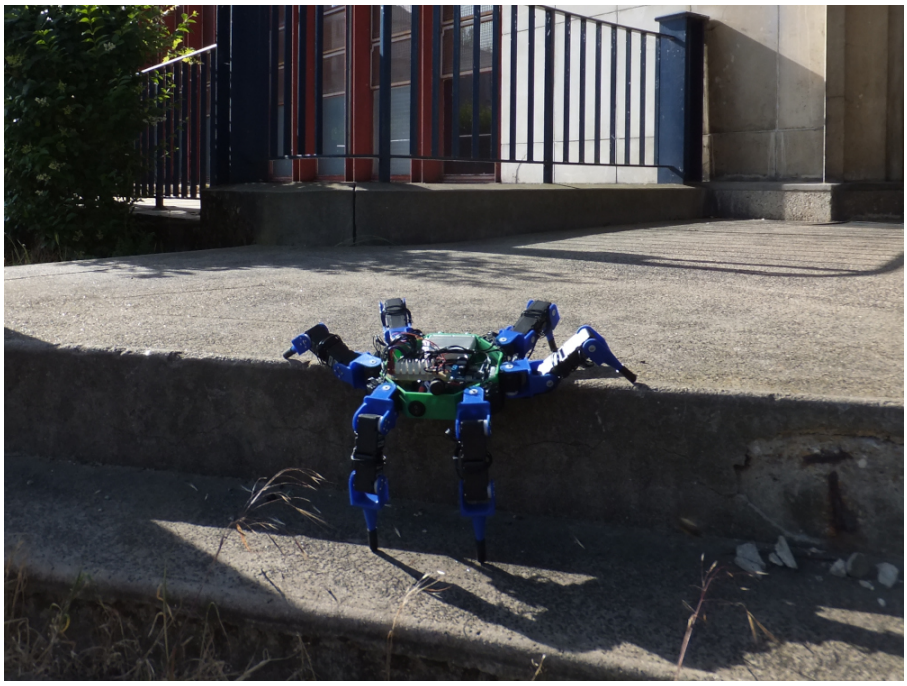

FIG. 2 : Robot hexapode devant l'ancien bâtiment Léonard de Vinci de l'ENS Paris-Saclay

2 Prise en main du robot

2.1 Prise en main des servo-moteurs

Le robot utilise 18 servo-moteurs Herkulex DRS-0101. Ils sont bon marché tout en offrant des performances convenables pour un hexapode.

Il aurait pu être souhaitable pour faciliter l'apprentissage d'utiliser des actionneurs plus faciles à modéliser et permettant un contrôle assez rapide pour être asservis en couple. Ce choix a été réalisé par d'autres projets de robotique tel que l'actionneur « transparent » de la patte de Solo qui est constituée d'un moteur sans balais et d'une courroie [1].

2.1.1 Commande des servo-moteurs

Les 18 servo-moteurs sont branchés sur un unique port série[1] de vitesse 500 000 baud/s[2]. Des identifiants allant de 0 à 17 ont été assignés. Ces identifiants sont les mêmes dans le fichier décrivant la simulation du robot ce qui facilite le transfert de la simulation vers la réalité.

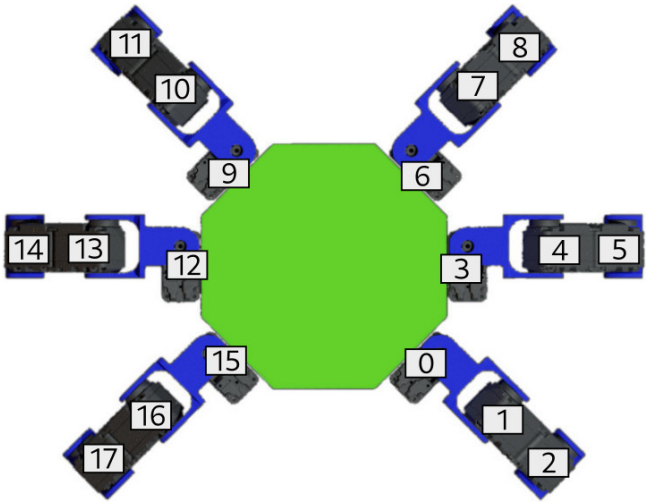

Fig. 3 : Assignation des identifiant des moteurs

La commande des 18 moteurs peut s'effectuer avec un unique paquet, sans attendre de réponse. Néanmoins la récupération des vitesses, positions et couples exercés sur les moteurs oblige d'envoyer 18 fois le même paquet et d'attendre 18 fois une réponse. C'est notre facteur limitant la rapidité de contrôle. **En pratique nous avons observé un temps de récupération des données entre 30 et 45 ms.** Il est donc raisonnable de commander avec une période de 50 ms les moteurs. Cette période étant grande, on devra utiliser l'asservissement en position interne au moteur plutôt d'envoyer des commandes en couple.

[1] La carte offre 2 ports séries, il peut être envisageable de faire 2 groupes de 9 moteurs.
[2] La vitesse maximale des moteurs est 667 000 baud/s mais cela n'est pas supporté par la carte embarquée.

Les moteurs sont accessibles sur le port série `/dev/ttyS4` avec par exemple PySerial. En pratique pour faciliter le développement, nous avons mis en place un proxy entre un port TCP ou UDP et le port série avec `socat`. La configuration est disponible sur la documentation : https://kraby.readthedocs.io/en/latest/use_motors/.

Il a été aussi question de mettre en place une classe Python permet d'abstraire le fonctionnement du moteur dans le cas d'un contrôle en position. Cette classe implémente une méthode pour envoyer un paquet à un moteur en calculant sa somme de contrôle, puis des méthodes pratiques pour commander et recevoir les angles et vitesses. Pour cette étape il a été question d'extraire un certain nombre de données de la documentation lacunaire de Herkulex et de tester des combinaisons de paramètres.

2.2 Prise en main de la centrale inertielle

Un travail de mise en place d'une centrale inertielle a été effectué au début du stage dans le but de pouvoir à termes transférer l'apprentissage de l'ensemble du robot vers la réalité.

La centrale inertielle a pour but de fournir l'orientation de la base du robot.

Lors de la réalisation du robot, une centrale inertielle **InvenSense MPU9250** a été choisie. Cette centrale a **l'avantage d'inclure un magnétomètre** ce qui permet d'éviter de dériver sur la direction du robot en ayant une référence magnétique. Cela permet alors au robot de garder son cap.

En réalité la MPU9250 n'est qu'une MPU6050 avec un magnétomètre AK8975 en sous-périphérique I2C.

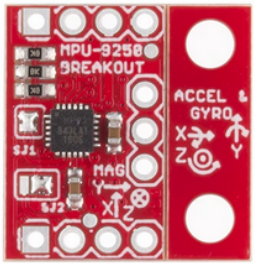

Fig. 4 : Centrale inertielle MPU9250 sur un support de prototypage

2.2.1 Compilation d'un noyau Linux avec le module inv-mpu6050

Il est possible d'utiliser la centrale inertielle en manipulant le bus I2C dans l'espace utilisateur[3] mais il existe déjà un module noyau Linux inv-mpu6050 permettant d'abstraire le fonctionnement et obtenir un périphérique virtuel donnant directement les données brutes de la centrale.

[3] L'espace utilisateur (userspace) est séparé de l'espace noyau (kernelspace).

Ce périphérique virtuel suit le standard du sous-système Linux Industrial I/O (iio) développé par Analog devices. Cela permet donc d'utiliser la centrale inertielle avec le module iio de Python ou avec un serveur iiod diffusant les données brutes sur le réseau. Il se trouve que iio est également utilisé industriellement sur des périphériques embarqués tels qu'un téléphone Android [2].

Le noyau pré-compilé de notre carte embarquée néanmoins ne contient pas ce module. On a alors deux possibilités : essayer de construire uniquement ce module puis de le charger dynamiquement après démarrage, ou de recompiler l'ensemble du noyau avec ce module. Dans les deux cas cela nécessite une chaîne de compilation ARM64 et les sources du noyau Linux modifiées par le constructeur de la carte.

Les détails techniques sur la procédure de compilation du noyau sont donnés sur la documentation en ligne du projet : https://kraby.readthedocs.io/en/latest/install_nanopi_linux/.

Après avoir installé ce noyau, il est possible d'activer la diffusion continue de l'axe X de l'accéléromètre de la centrale inertielle par les commandes suivantes :

```
echo 1 > "/sys/bus/iio/devices/iio :device1/scan_elements/in_accel_x_en"
echo 1 > "/sys/bus/iio/devices/iio :device1/buffer/enable"
hexdump -C "/dev/iio :device1"
```

Un exemple de code Python pour récupérer l'ensemble des données brutes est disponible sur la documentation en ligne du projet : https://kraby.readthedocs.io/en/latest/use_mpu9250/.

Le fait d'utiliser le module IIO plutôt que de manipuler nous même le bus I2C permet de laisser le noyau gérer le signal d'interruption de la centrale et de stocker à une fréquence modifiable ses données dans une mémoire tampon.

2.2.2 Traitement des données

La centrale inertielle renvoie seulement des données brutes de ses capteurs. On a besoin d'**un algorithme de fusion de données** afin d'obtenir l'orientation de notre robot.

Un des algorithmes les plus couramment utilisés dans les périphériques embarqués est celui développé par Sebastian Madgwick et Robert Mayhony [3]. Il réalise la fusion de données en utilisant une représentation par des quaternions afin d'éviter le phénomène du blocage de cardan.

Il se trouve que le papier est accompagné d'un code C implémentant cet algorithme en ayant développé les calculs, donc sans utiliser une librairie de calcul de quaternions. Il y a deux fonctions, une « IMU » (Inertial Motion Unit) dans le cas d'un accéléromètre et gyromètre, et une « AHRS » (Attitude and Heading Reference System) dans le cas où on ajoute un magnétomètre.

3 Entraînements sur simulation

3.1 Conception de la simulation

La simulation est constituée d'un simulateur physique chargeant une description de notre robot.

3.1.1 Simulateur physique

Nous avons choisi d'utiliser **le simulateur BulletPhysics** sous licence zlib. Il est développé par Erwin Coumans (Google). BulletPhysics est une librairie souvent utilisée dans les simulations de robot ou les moteurs de jeux. Il est équivalent à Mujoco qui nécessite une licence payante, mais est un peu moins facile d'utilisation [4].

Un module Python a été développé pour pouvoir facilement l'utiliser. Voici un code qui va charger notre environnement avec des moteurs statiques :

```
import pybullet as p

p.connect(p.GUI)  # Open new physic server with GUI
p.setGravity(0, 0, -9.81)  # We are still on earth
p.setAdditionalSearchPath(getDataPath())  # Add pybullet_data
p.loadURDF("plane.urdf")  # Load a ground
p.loadURDF("hexapod.urdf")  # Load the full robot
```

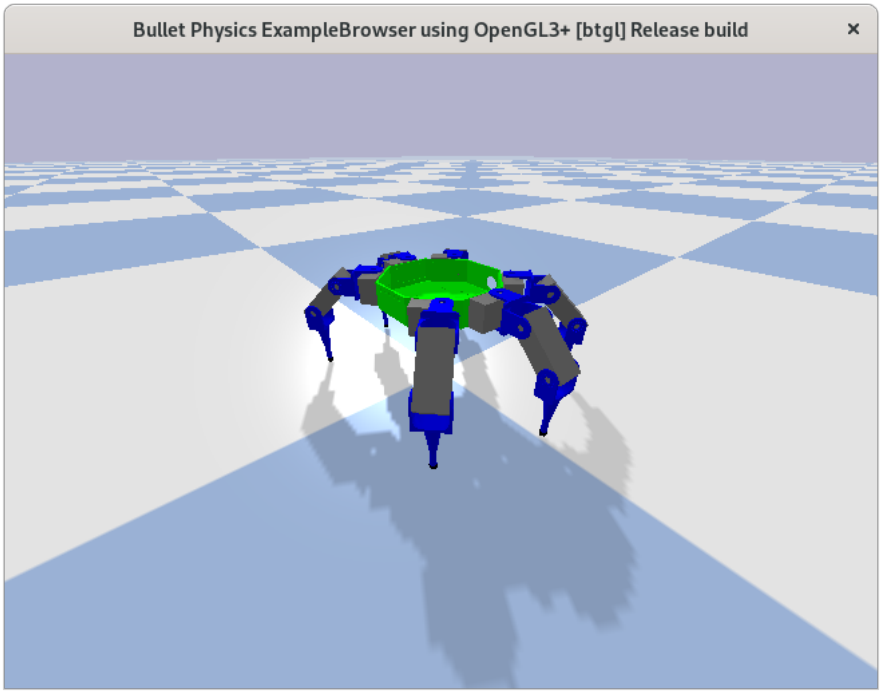

FIG. 5 : Description d'un sol et d'un robot hexapode chargés dans PyBullet

3.1.1.1 Problème de glissement. Initialement, charger le robot et le laisser sur une position neutre sur ses pattes provoquait un **glissement non réaliste** du robot. Nous avons découvert qu'il faut activer le « `friction_anchor` » dans la description du robot pour activer **un correcteur cartésien de la position** et éviter que le calcul mécanique diverge.

3.1.2 Description du robot

Pour réaliser la description du robot il a été question d'extraire différentes données de la modélisation 3D du modèle et de la documentation technique des composants puis d'**écrire un fichier XML**.

La description du robot a été initialement réalisée dans le format SDFormat, associé au logiciel Gazebo qui permet de simuler un environnement en utilisant un simulateur physique tel que BulletPhysics. Néanmoins il s'est avéré que ce format n'est pas aussi mature que le format URDF développé par le projet Robotic Operating System (ROS). Cela empêchait alors de faire une description complète du robot en profitant de fonctionnalités avancées de BulletPhysics. De plus nous avons rencontré des inconsistances entre le chargement des liaisons mécaniques d'un SDFormat dans Gazebo et dans d'autres logiciels. Nous avons ainsi choisi de réécrire **la description en URDF** (Unified Robot Description Format).

Pour gagner du temps et éviter les erreurs, nous utilisons un modèle dans le format Jinja2 qui permet de générer le fichier URDF final. Ainsi nous pouvons itérer sur les composants, inclure des fichiers, calculer les angles et distances et générer le code XML correspondant. Il suffit alors d'un simple script Python pour compiler ce modèle et générer la description d'une seule patte ou de tout le robot.

La description complète du robot est disponible ici : https://github.com/erdnaxe/kraby/blob/master/gym_kraby/data/hexapod.urdf.

3.1.2.1 Mesure des inerties La description URDF contient les inerties des différentes pièces.

Pour le cas d'une pièce imprimée en 3D, **nous avons utilisé MeshLab pour calculer son inertie numériquement**, tout en faisant l'hypothèse que la pièce imprimée est remplie de façon homogène. Pour cela chaque pièce a été exportée en format STL, puis importée dans MeshLab. Avec l'outil « *Compute Geometric Measures* » on obtient une matrice d'inertie.

Néanmoins Meshlab considère la densité de notre matériau unitaire. On réalise donc un script Python pour convertir la matrice d'inertie dans le format XML final.

```
mass = float(input("Object Mass is  : "))
volume = float(input("Mesh Volume is  : "))
print("Inertia Tensor is  :")
j = [list(map(float, input("  ").strip().split(" "))) for _ in range(3)]

for i in range(3) :
    for k in range(3) :
        # Change density and convert millimeter to meter
```

```
        j[i][k] *= mass/volume*0.000001

print(f"""\n<inertia ixx="{j[0][0] :.16f}" ixy="{j[0][1] :.16f}" ixz="{j
    [0][2] :.16f}" iyy="{j[1][1] :.16f}" iyz="{j[1][2] :.16f}" izz="{j[2][2] :.16
    f}" />""")
```

Pour le cas des autres pièces telles que les servo-moteurs, nous les avons considérées comme des pavés homogènes. Si nous observons des écarts conséquents entre la simulation et la réalité nous pourrons revenir plus tard sur cette hypothèse.

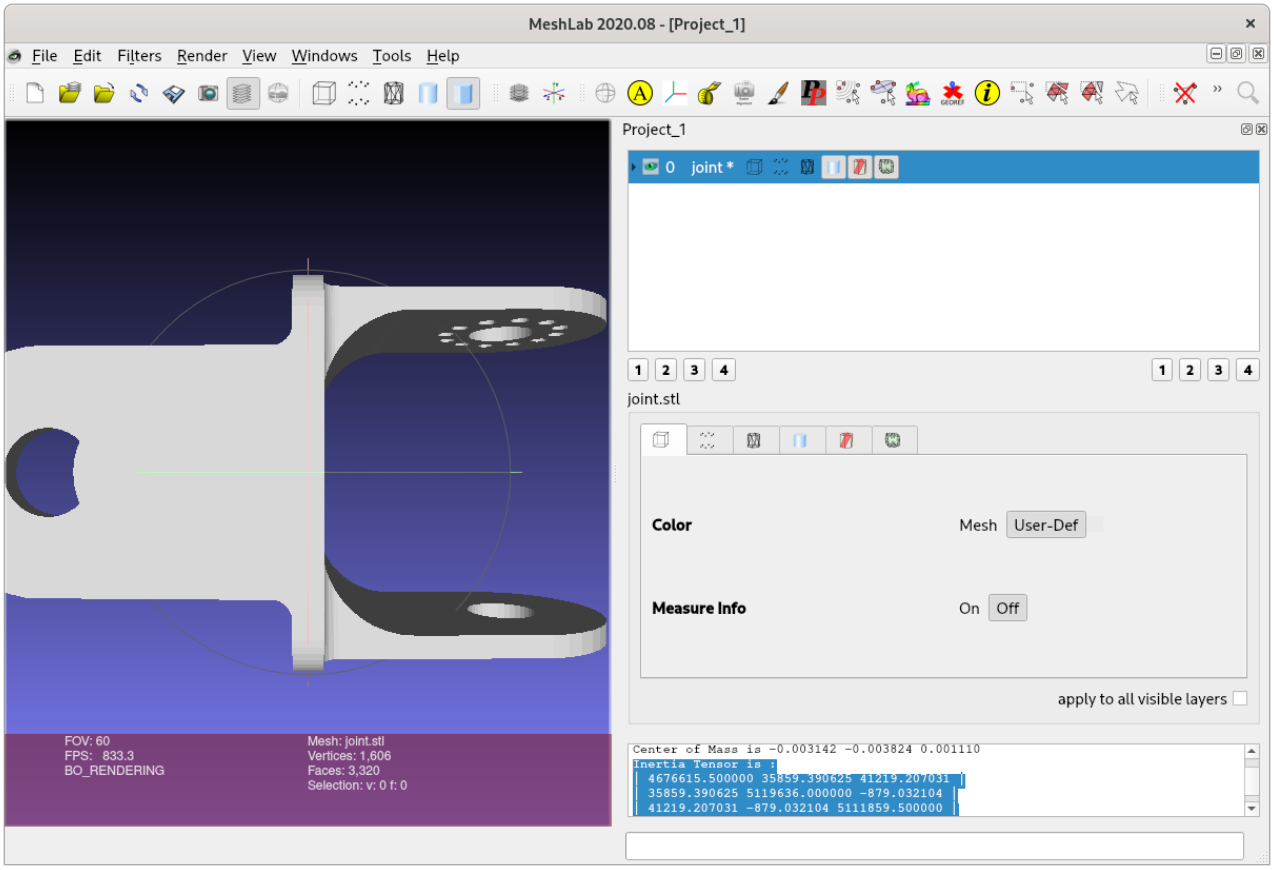

FIG. 6 : Mesure de l'inertie dans Meshlab

3.1.2.2 Simulation des moteurs Les moteurs correspondent à des liaisons pivots que l'on contrôle en vitesse.

On définit et commande ces moteurs avec la méthode `setJointMotorControl` de PyBullet. On précise alors le couple maximal de cette liaison.

Pour ce qui est de la simulation des frottements, nous définissons tout cela dans la liaison dans le fichier URDF. Nous avons pour le moment désactivé les frottements pour les premiers essais. Il faudra réadapter ces paramètres une fois l'entraînement fonctionnel.

```
<joint name="servomotor_0" type="revolute">
  <origin xyz="0 0.011 0" rpy="0 0 0" />
  <parent link="servomotor_0" />
  <child link="joint_0" />
  <axis xyz="0 0 1" />
  <dynamics damping="0" friction="0" />
  <!-- Herkulex  : 361.4°/s i.e. 6.308rad/s, 12kg.cm i.e. 1.18Nm -->
  <limit effort="1.18" velocity="6.308" lower="-0.46" upper="0.46" />
</joint>
```

Le modèle du moteur est potentiellement trop simplifié pour réaliser par la suite le passage à la réalité. PyBullet intègre également un modèle plus complexe de moteur sur lequel on peut régler les coefficients de l'asservissement [5].

3.2 Mise en place des environnements d'apprentissage

Nous avons choisi d'implémenter nos environnements d'apprentissage en un module Python en suivant la convention initiée par OpenAI Gym [6].

3.2.1 Rapide introduction à l'apprentissage par renforcement profond

L'apprentissage par renforcement profond consiste à utiliser des techniques d'apprentissage par réseau de neurones pour prendre des décisions et ainsi contrôler par exemple un robot. Plus précisément, le but est qu'**un agent** apprenne à maximiser **le gain collecté total** (return) en interagissant dans **un environnement**.

À chaque pas temporel de l'environnement, l'agent décide d'**une action** à partir de **l'observation** en suivant **sa politique**. Ainsi le but du renforcement est d'**optimiser cette politique** pour que l'agent prenne les meilleurs choix.

Pendant l'entraînement, l'environnement va également renvoyer **un gain** associé à l'action que l'agent vient d'effectuer. La somme de tous ces gains constitue **le gain collecté total** (return) de l'épisode.

3.2.2 Environnement au format OpenAI Gym

OpenAI Gym est un module Python qui va définir **une interface standard pour créer des environnements réutilisables**. De plus ce module contient de nombreux exemples pour pouvoir tester et comparer des algorithmes d'apprentissage.

L'exemple `Reacher-v2` fourni dans le module est une bonne base de comparaison car il simule un bras dans un espace 2D qui a pour but d'atteindre un objectif. Dans notre cas on a une structure de bras différente avec une patte, et on est dans un espace 3D. De plus nous n'utilisons pas Mujoco mais BulletPhysics.

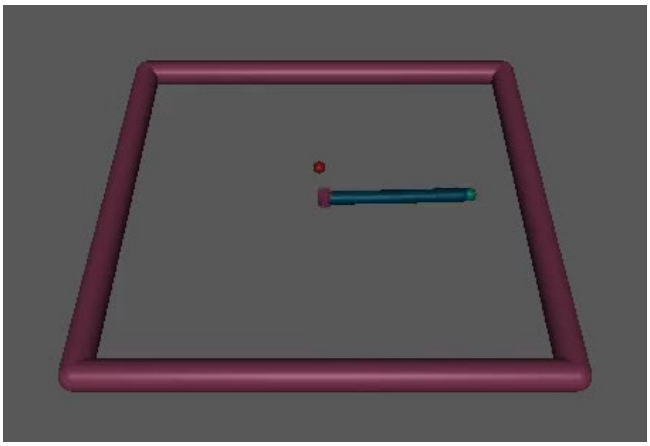

FIG. 7 : Environnement `Reacher-v2`, le bras bleu peut se plier et le but est de minimiser la distance entre la boule verte et rouge

Un environnement Gym est une classe Python implémentant au moins un ensemble de méthodes pré-définies.

```python
import gym

class ExampleEnv(gym.Env) :
    def __init__(self, param1=0.01, param2=False, param3=200) :
        super().__init__()
        # Initialize the environment

    def reset(self) :
        # Reset the environment
        return observation

    def step(self, action) :
        # Do an action
        return observation, reward, done, {}

    def render(self, mode='human') :
        # Render the environment for human visualization or for recording

    def close(self) :
        # Close environment

    def seed(self, seed=None) :
        # Change the random number generator seed
```

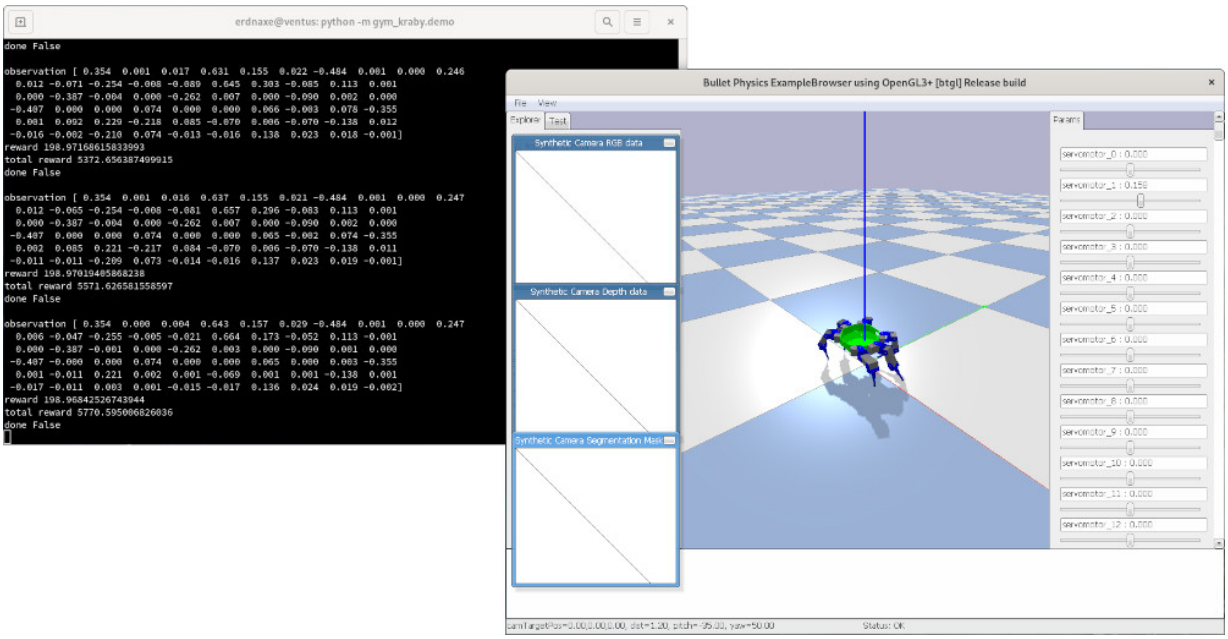

Fig. 8 : Environnement OpenAI Gym pour simuler l'ensemble du robot

Dans le cas des environnements où un objectif doit être atteint, nous ajoutons une croix de débogage pour visualiser l'objectif.

3.2.2.1 Normalisation des entrées et sorties.

Pour rendre l'apprentissage plus facile, nous normalisons l'espace d'action et d'observation entre -1 et 1. Une action de -1 ou 1 sur un moteur correspondra alors à la vitesse maximale. Une observation de -1 ou 1 correspondra à la valeur maximale possible.

3.2.3 Publication du module Python

Ce module Python est publié sur Pypi et ainsi facilite l'installation de la simulation et des environnements : https://pypi.org/project/gym-kraby/.

En plus de la publication du paquet Python, les sources de l'ensemble du robot sont distribuées sous une licence MIT permissive sur GitHub : https://github.com/erdnaxe/kraby/.

Ainsi il suffit d'un environnement au moins Python 3.7 et de lancer les commandes suivantes pour simuler le robot et avoir les environnements OpenAI Gym.

```
pip3 install gym_kraby --user
python3 -m gym_kraby.demo
```

Le module Python inclut 4 environnements :

- **HexapodBulletEnv-v0** : simule le robot hexapode en entier,
- **HexapodRealEnv-v0** : commande le robot hexapode en entier,
- **OneLegBulletEnv-v0** : simule une seule patte du robot,
- **OneLegRealEnv-v0** : commande une seule patte du robot.

Outils de développement. Nous avons utilisé **Git** avec le service GitHub pendant ce stage pour versionner notre code et suivre les erreurs à corriger. De plus nous avons mis en place un **environnement de déploiement continu** afin d'automatiser la création du paquet Python et d'une image Docker. Ainsi à chaque nouvelle version du code poussée vers le serveur Git, un paquet Python et une image Docker correspondant sont créés.

Pour garantir une bonne qualité du code et vérifier l'existence d'erreurs de syntaxe, nous avons également utilisé le service Codacy qui permet d'analyser le code avec des outils adaptés tel que Flake8 et Remark.

3.2.4 Période d'échantillonage des environnements

Pour interagir avec l'environnement, il faut régulièrement envoyer une action avec la méthode `step` qui va renvoyer un nouveau vecteur d'observation et un gain.

Dans le cas d'une simulation, on peut choisir l'intervalle de temps que l'on souhaite. Néanmoins lorsque l'on applique cela au robot réel on est limité par la vitesse de collecte des données des moteurs pour construire le vecteur d'observation. **Dans notre cas, l'intervalle de temps devra être supérieur à 50 ms.**

Un problème se pose néanmoins si on exécute un pas de simulation tous les 50 ms. La simulation physique va alors également effectuer des pas toutes les 50 ms ce qui va l'**écarter d'un comportement réaliste**. Pour pallier à ce problème, nous faisons faire **12 pas de simulation physique d'intervalle 4.2 ms tous les pas de l'environnement (de 50 ms)**.

3.2.5 Conception d'une fonction de gain

Une des étapes les plus délicates en apprentissage par renforcement est de concevoir une fonction qui calcule un gain après une action.

Dans le cas d'une patte, le gain utilisé correspond à l'opposé du carré de la distance entre l'extrémité de la patte et l'objectif. Ainsi le gain total sur un épisode est toujours négatif.

Il aurait aussi pu être possible de définir le gain comme la progression vers l'objectif. Ainsi il serait positif si on approche de l'objectif ou négatif si on s'en éloigne.

On pourrait ajouter d'autres mesures pondérées dans cette fonction, telle que l'orientation de la patte pour favoriser une patte verticale ou l'énergie consommée pour le déplacement. Néanmoins nous n'avons pas exploré ces modifications.

Il est aussi possible d'interrompre plus tôt un épisode de simulation si le robot se comporte mal. Néanmoins dans ce cas il ne faut pas que le gain soit négatif, sinon le robot apprendra à finir l'épisode le plus vite.

3.3 Mise en place de l'algorithme d'apprentissage

Dans le cadre de ce stage nous avons choisi d'utiliser l'algorithme **Proximal Policy Optimization** (PPO) développé par l'équipe de recherche d'OpenAI [7].

D'autres algorithmes qui pourraient être envisagés sont :

- **Deep Q-learning** (DQN) qui marche bien sur des environnements avec un espace d'action discret. DQN marche généralement moins bien sur les environnements à contrôle continu car la discrétisation naïve de l'environnement continu subit la malédiction de la dimension (curse of dimensionality) [8].

- **Une descente de gradient classique sur la politique** (VPG pour Vanilla Policy Gradient) qui a une efficacité par échantillon plus faible et n'est pas forcément aussi robuste. Cette algorithme met à jour les paramètres de la politique par une descente de gradient stochastique sur la performance de cette politique [9].
- **Les méthodes Trust Region / natural Policy gradient** (par exemple TRPO) qui ont une efficacité similaire à PPO tout en utilisant un algorithme un peu plus compliqué [7].

3.3.1 On-policy vs Off-policy

Un algorithme on-policy optimise la politique qui est utilisée pour générer les données. **Un algorithme off-policy** optimise une politique différente de celle qui a été utilisée pour générer les données. L'off-policy est particulièrement intéressant si on a en avance un dataset de comportements et que l'on souhaite apprendre dessus.

Dans notre cas PPO est on-policy et memory-less. On ne réutilise pas les anciennes données. Cela veut dire qu'un batch de données générées par simulation ne sera utilisé que pour entraîner la prochaine politique. Un algorithme on-policy a une efficacité par échantillon naturellement plus faible que de l'off-policy.

À première vu cela est problématique car avec notre faible dimensionnalité on se retrouve à sous-utiliser le GPU pour l'apprentissage et à être limité par la génération des données sur CPU.

Néanmoins en contre-partie, cela augmente la stabilité de l'entraînement en optimisant directement la politique utilisée pour générer les données suivantes. Le déficit sur l'efficacité a néanmoins été réduit d'année en année avec VPG, puis TRPO, puis PPO [9].

3.3.2 Les implémentations existantes de PPO

Il existe de nombreuses implémentations de PPO. Nous n'avons pas tenté de réimplémenter l'algorithme avec PyTorch ou Tensorflow 2. Néanmoins il existe une implémentation de PPO en PyTorch dans OpenAI SpinningUp qui est destinée à des fins pédagogiques plutôt qu'aux performances.

- **pytorch-a2c-ppo-acktr-gail** contient une implémentation en PyTorch de PPO tel que décrit par le papier d'origine. Les premiers tests ont été effectués avec cette librairies car PyTorch était plus facile à mettre en place. Néanmoins cette librairie manque de documentation et n'est pas destinée à un but pédagogique.
- **OpenAI Baselines** contient l'implémentation officielle de PPO en TensorFlow1. Néanmoins contrairement à ce que l'on pourrait imaginer, la version de PPO implémentée contient de nombreuses modifications non documentées dans le papier permettant d'avoir de meilleurs performances. Le code utilise des conventions hétérogènes, difficiles à lire et à prendre en main.

- **StableBaselines** est une modification de OpenAI Baselines afin de rendre cette librairie plus lisible et plus facile d'accès à des débutants [10]. **Nous avons au final utilisé cette librairie pour l'ensemble du projet.**
- **Garage** implémente PPO pour TensorFlow1 et PyTorch, mais la documentation bien que complète, n'est pas destinée à des débutants.
- **TensorForce** implémente PPO pour TensorFlow2 et permet d'utiliser de nombreuses fonctionnalités tel qu'un analyseur de performance et un optimiseur d'hyperparamètres. Néanmoins il n'était pas encore stable en TensorFlow2 au début du stage.
- **RLlib** implémente PPO en PyTorch. L'implémentation semble propre et la documentation est bien écrite. Nous n'avons néanmoins pas eu l'occasion de tester cette librairie.

À noter que RLlib, et d'autres systèmes d'entraînement comme Facebook ReAgent ou Intel Coach proposent d'effectuer l'entraînement sur des noeuds de calculs CPU et GPU séparés. Ainsi les noeuds CPU génèrent les données d'entraînement et les noeuds GPU effectuent l'entraînement.

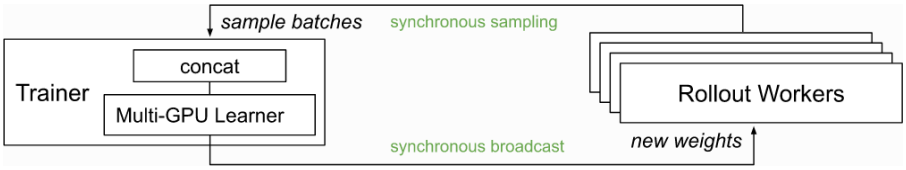

FIG. 9 : Architecture distribuée pour PPO dans RLlib (extraite de la documentation de RLlib)

Pour donner un ordre de grandeur, **OpenAI Five** (premier agent qui a battu le champion mondial de Dota 2) a été entraîné avec 128 000 noeuds CPU et 256 noeuds GPU (Nvidia P100) sur Google Cloud Plateform [11]. On remarque que OpenAI utilise 500 CPU pour générer assez de données pour 1 GPU. On verra par la suite que sur nos environnements on est également limité par le nombre de cœurs CPU.

3.4 Entraînement d'une patte

Le but de cette partie est de montrer l'effet de chaque hyperparamètre sur l'apprentissage. Pour cela on va partir des hyperparamètres de `ReacherBulletEnv-v0` optimisés pour Stable-Baselines [12] comme base et les adapter :

- `n_envs=32` simulations en parallèle,
- `n_timesteps=250e3` pas de temps total (condition d'arrêt de l'entraînement),
- `policy='MlpPolicy'`, 2 couches de 64 neurones,
- `n_steps=128` pas de temps par simulation, soit 4 épisodes de 32 pas de temps,
- `nminibatches=32`, on découpe un batch de données en 32,
- `lam=0.95`, facteur de lissage de l'estimateur de la fonction d'avantage,
- `gamma=0.90`, facteur de réduction du gain,
- `noptepochs=30`, nombre d'époques d'optimisation,
- `ent_coef=0.01`, facteur de pondération de l'entropie,
- `learning_rate=10e-4`, rythme d'apprentissage,
- `cliprange=0.2`, plage de saturation d'une mise à jour de la politique.

Il faut savoir qu'il existe des outils pour réaliser les étapes suivantes automatiquement, par exemple Optuna [13] ou un outil intégré dans TensorFlow2. Mais cela nécessite beaucoup de ressources de calcul et notre but est plutôt d'expliquer les effets des paramètres pour mieux comprendre ce qui se passe.

3.4.1 Comparaison de différents vecteurs d'observation

Lorsque l'on construit l'environnement OpenAI, il faut définir le contenu du vecteur d'observation que l'on donne en entrée à l'agent. Il y a rarement une unique combinaison qui marche. Il faut trouver le bon compromis entre ne pas garder trop d'observations ce qui ralentirait l'apprentissage et enlever trop de données qui sont nécessaires à l'apprentissage.

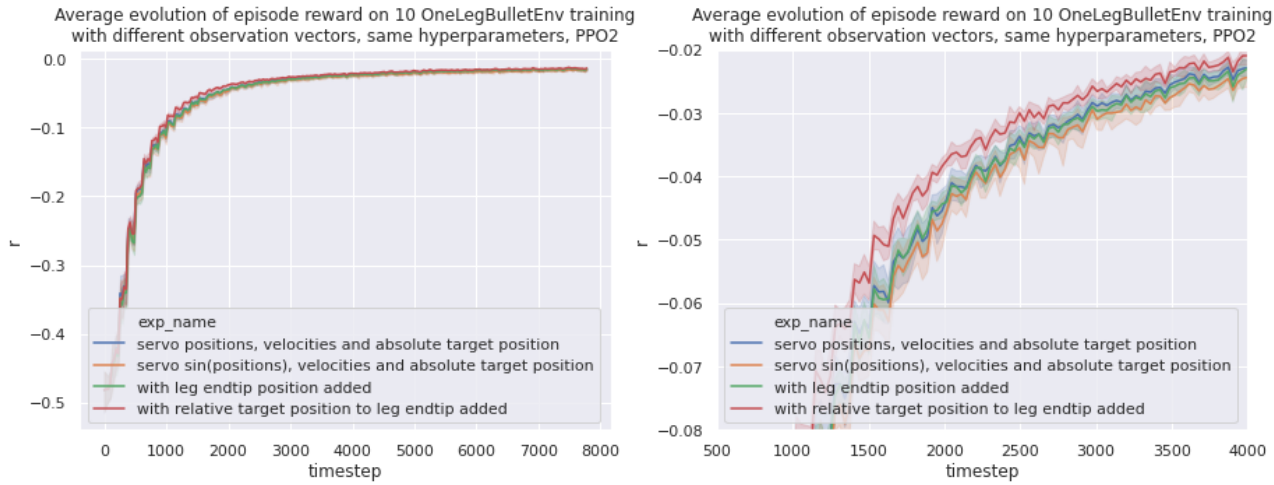

FIG. 10 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement, avec différents vecteurs d'observation

`ReacherBulletEnv-v0` utilise des cosinus et sinus pour les observations des positions angulaires. Dans notre cas, on n'a besoin que du sinus parce que nos moteurs vont de $-\pi/2$ à $\pi/2$. On observe que remplacer l'angle par le sinus de l'angle n'aide pas l'entraînement (courbe jaune).

Une observation intéressante est que l'on obtient des performances d'apprentissage similaires avec ou sans le vecteur indiquant la position absolue du bout de la patte. Cela implique que l'on n'aura pas à implémenter la mécanique directe de la patte au moment de transférer la politique de contrôle de la patte sur le vrai robot. **L'agent a appris à contrôler les angles de la patte pour atteindre l'objectif.**

Également il ressort de cette expérience que modifier le vecteur d'observation n'affecte pas autant les performances qu'optimiser les hyperparamètres. C'est bien plus intéressant de passer son temps à optimiser les hyperparamètres que le contenu de ce vecteur.

Pour tous les prochains entraînements on utilisera alors le vecteur d'observation suivant :

$$obs = \begin{pmatrix} \theta_1 \\ \omega_1 \\ \theta_2 \\ \omega_2 \\ \theta_3 \\ \omega_3 \\ target_x \\ target_y \\ target_z \end{pmatrix}$$

θ_i étant l'angle normalisé entre -1 (-90°) and 1 (90°) et ω_i la vitesse normalisée.

3.4.2 Optimisation du nombre d'environnements à lancer en parallèle

Au début de l'entraînement, on instancie un certain nombre d'environnements en mémoire que l'on utilise ensuite pour générer les données d'apprentissage. Chaque environnement contient un monde simulé par BulletPhysics avec une seule patte du robot. Avec un unique environnement (`n_envs=1`), la génération des données n'utilisera que 2 cœurs CPU et sera lente.

On observe sur un processeur i7-8750H (6 cœurs physiques, 12 cœurs logiques) que `n_envs=12` ne permet pas de tirer le plus de performance du processeur. Augmenter le nombre au-delà accélère encore un peu plus l'apprentissage.

Néanmoins un environnement utilise aussi de la mémoire vive. Une règle simple est alors d'augmenter le nombre d'environnement jusqu'à que l'on utilise bien le processeur et que l'on ne dépasse pas la quantité de mémoire vive de la machine. `n_envs=32` nécessite environ 12 GiB de mémoire vive, ce qui laisse un peu de marge pour les autres programmes sur un ordinateur ayant 16 GiB de mémoire.

3.4.3 Optimisation de la taille de batch

La taille de batch représente le nombre de steps simulés utilisés pour effectuer une époque d'entraînement. Parce que l'entraînement est réalisé sur GPU et que l'on est en faible dimensionnalité, augmenter la taille de batch devrait mieux utiliser le GPU et ainsi améliorer la vitesse d'entraînement.

À noter que la position initiale et l'objectif de la patte sont tirés aléatoirement. On a donc intérêt à avoir un batch suffisamment grand pour avoir assez de variance dans les données.

Dans la dernière partie, nous avons fixé `n_envs`. On ne peut donc maintenant changer la taille de batch qu'en modifiant le nombre d'épisodes simulés dans chaque environnement, ou le nombre de minibatchs.

$$batch\ size = \frac{n_{steps} \times n_{envs}}{n_{minibatches}}$$

3.4.3.1 Optimisation du nombre de minibatchs

Avec `nminibatches=1` on n'a pas de division du batch de données. On réalise alors **une descente de gradient avec l'ensemble du batch**. Augmenter `nminibatches` permet de découper les données générées par les simulations. On réalise alors **une descente de gradient avec des mini batch**.

Avoir `nminibatches` plus grand que 1 devrait diminuer les performances dans notre cas car on n'apprend pas sur beaucoup de données.

Les entraînements suivants ont été réalisés avec `n_envs=32` et `n_steps=128`. **Donc avec le nombre minimal de mini batchs on obtient la taille maximale de 4096 pas de temps dans un batch de données.**

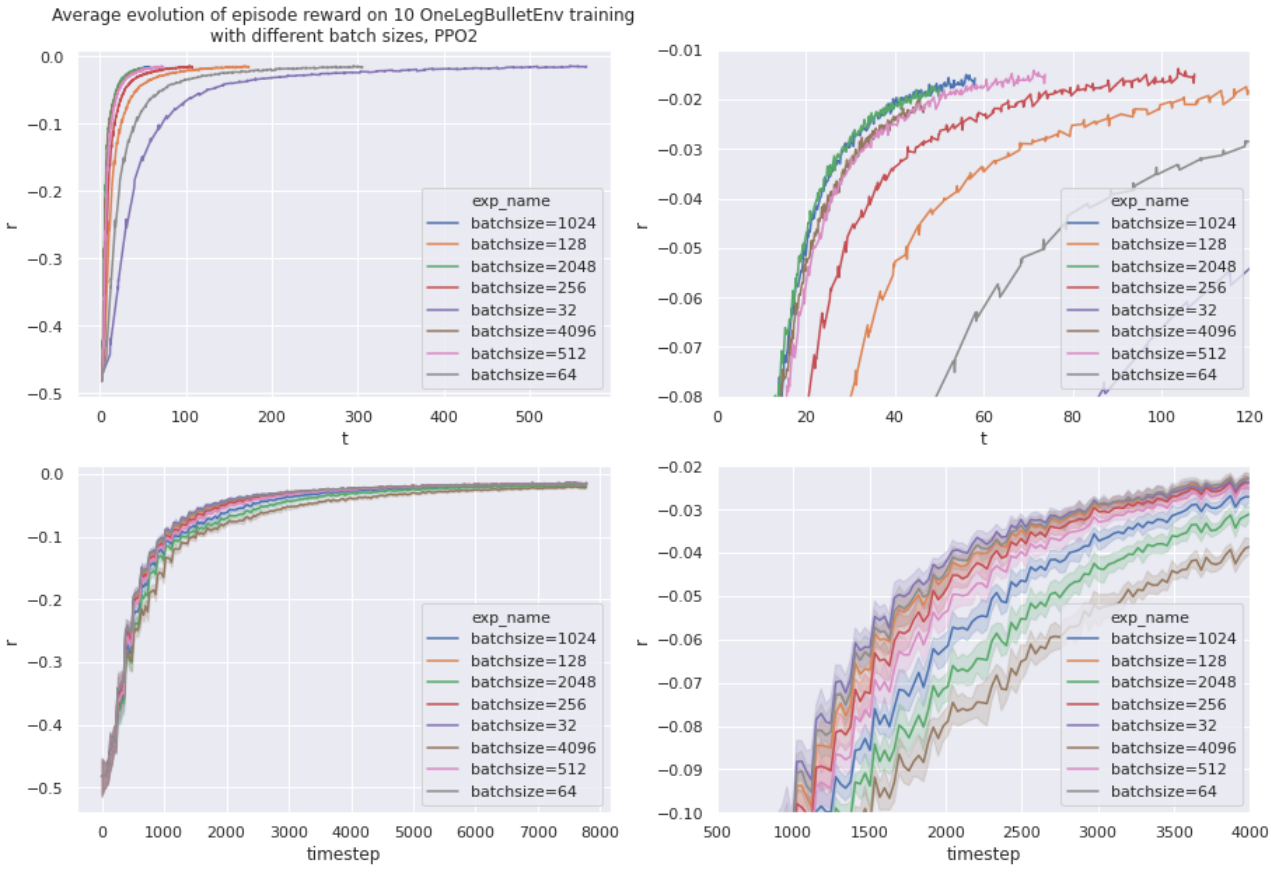

FIG. 11 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement ou du temps réel, avec différents nombre de mini batchs

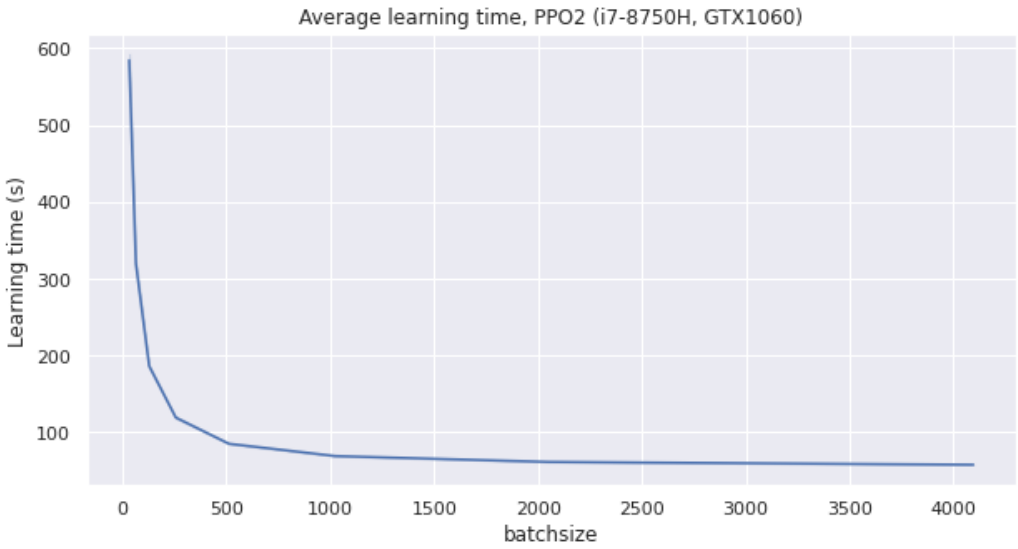

FIG. 12 : Évolution du temps d'entraînement total en fonction de la taille d'un batch de données (en changeant le nombre de mini batchs)

Découper un batch de données en mini batchs permet de légèrement améliorer l'efficacité de l'entraînement sur les données générées, mais en contre-partie cela ralentit grandement la vitesse totale de l'entraînement car on réduit encore plus nos dimensions sur GPU.

La conclusion de cette expérience est que l'on peut garder `nminibatch=1` c'est-à-dire ne pas séparer les données générées lors de l'apprentissage. Ainsi la taille de batch ne sera décidée que par le nombre d'environnements lancés en parallèle multiplié par le nombre de pas de temps effectués dans chaque environnement.

3.4.3.2 Optimisation du nombre d'épisodes de simulation Un épisode de simulation contient 32 pas de temps, donc $n_{steps} = 32 \times n_{episodes}$. Le but ici est de changer le nombre d'épisodes simulés et donc au final le nombre de pas de temps dans les données. Plus d'épisodes de simulation utilisera plus de temps CPU et créera un batch de données plus large pour la prochaine optimisation de la politique.

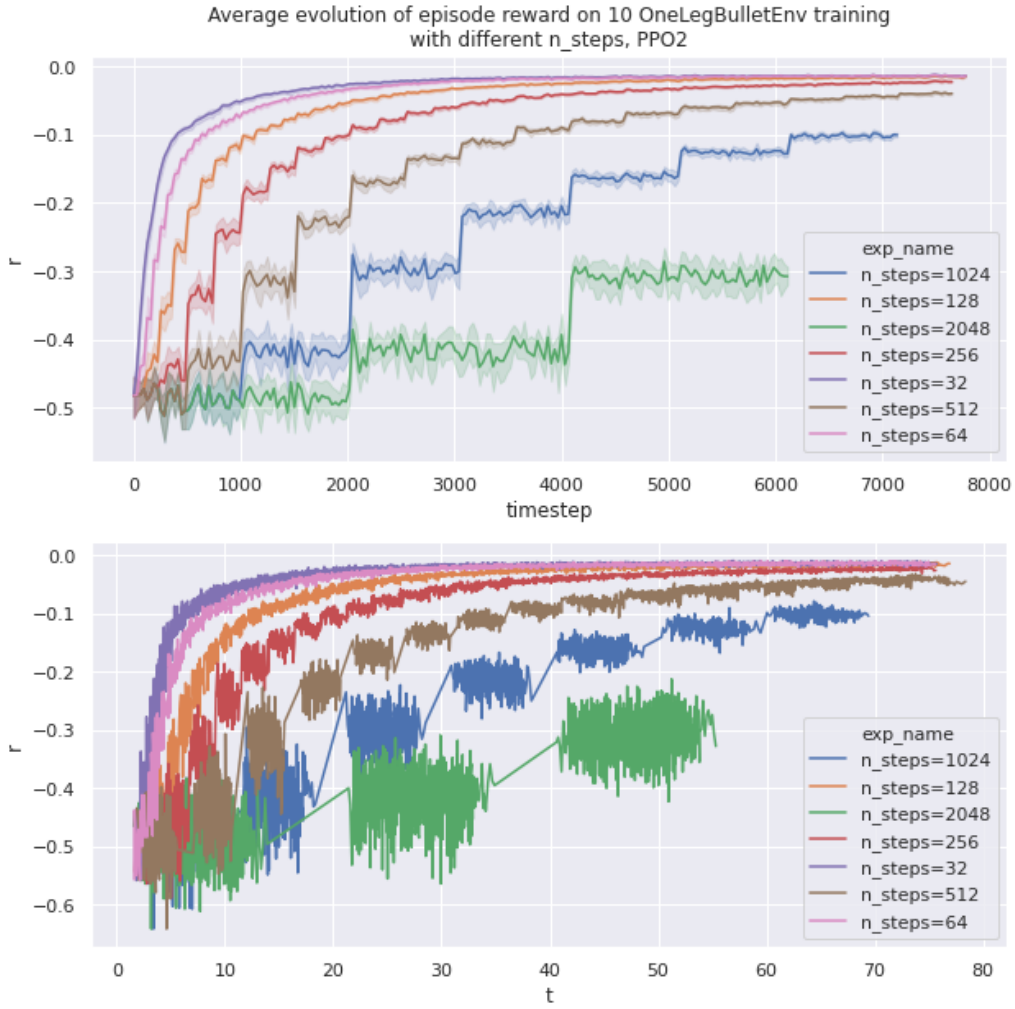

FIG. 13 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement ou du temps réel t, avec différents nombre de pas de temps simulés

Plus `n_steps` augmente, plus de temps GPU est nécessaire. Un batch de données trop large diminue les performances d'apprentissage et les données sont moins rentabilisées. On remarque également que ne simuler qu'un épisode dans les 32 environnements semble suffisant pour avoir assez de variance sur les données.

Quand `n_steps` est plus grand, il semblerait que le temps total d'apprentissage soit légèrement plus rapide. En réalité c'est parce que la condition de fin sur le nombre de pas de temps n'est pas un multiple du nombre de pas dans un batch de données. L'apprentissage pourrait aussi être légèrement plus rapide car moins d'aller-retour CPU-GPU sont nécessaires.

3.4.4 Optimisation du nombre d'époques d'optimisation

Le nombre d'époques d'optimisation (`noptepochs`) correspond au nombre de fois que l'on utilise les données générées pour optimiser les poids du réseau. Augmenter le nombre d'optimisations utilise plus de GPU mais en contre-partie rentabilise plus les données simulées. Si trop d'optimisations sont réalisées, le processus d'apprentissage sera plus lent.

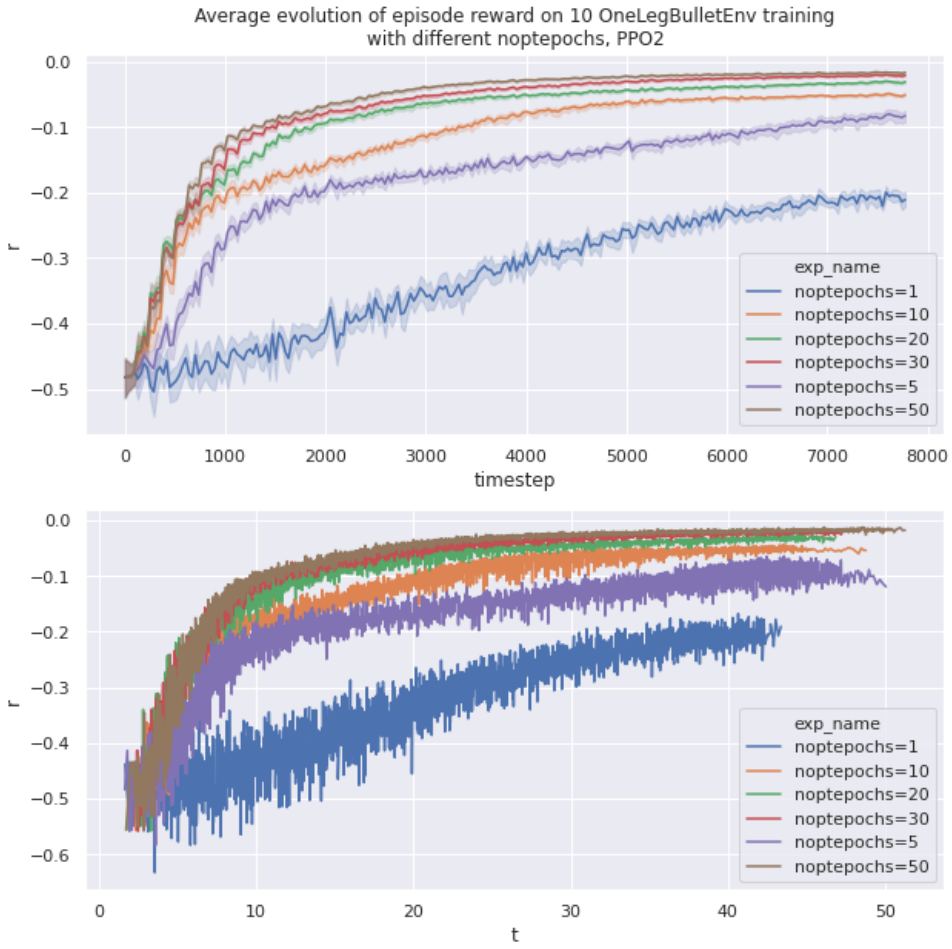

FIG. 14 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement ou du temps réel t, avec différents nombre d'époques d'optimisation

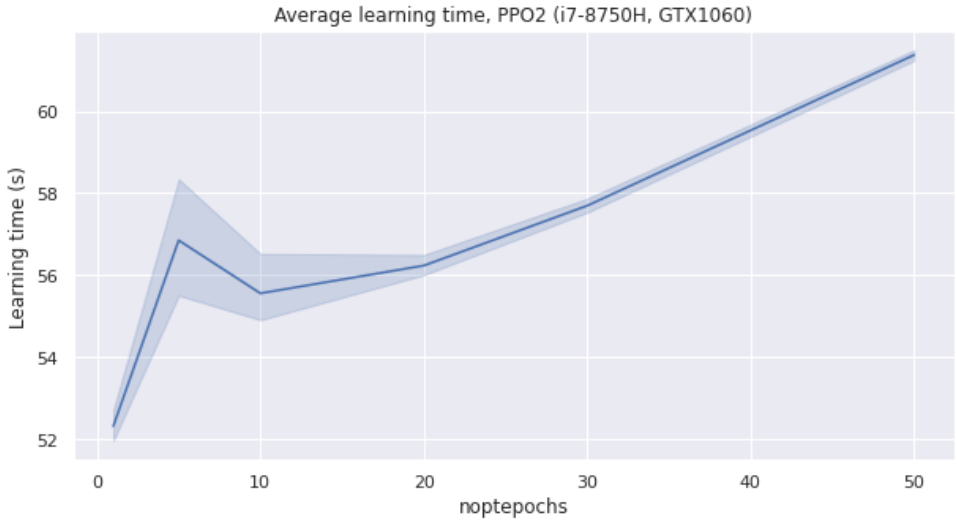

FIG. 15 : Évolution du temps total d'apprentissage en fonction du nombre d'époques d'optimisation

On observe comme prévu qu'augmenter `noptepochs` permet d'améliorer l'efficacité de l'apprentissage mais avec un temps total plus lent. Un bon compromis semble être `noptepochs =30`.

3.4.5 Optimisation du facteur de réduction (discount factor)

Le facteur de réduction de gain (discount factor, `gamma`, γ) est un coefficient utilisé lors de la sommation de tous les gains d'un épisode de simulation.

$$return = \sum_{n} \gamma^n \, reward_n$$

$\gamma = 1$ signifie que l'entraînement ne favorisera pas une politique permettant d'atteindre l'objectif plus rapidement. On retrouve souvent dans la littérature des valeurs entre 0,9 et 0,999.

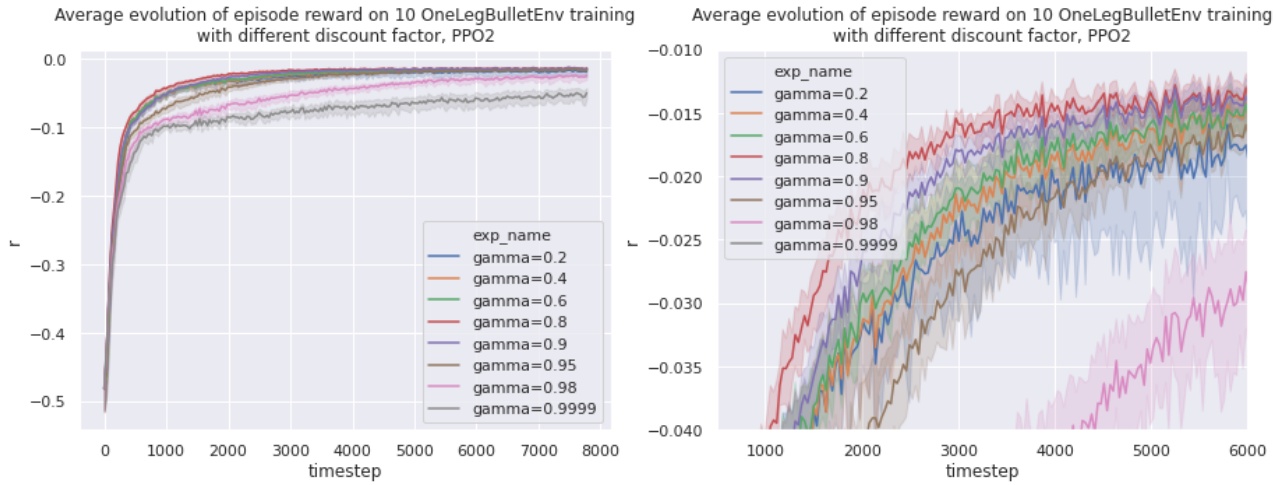

FIG. 16 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement ou du temps réel t, avec différents facteurs de réduction

On observe qu'un facteur γ entre 0,8 et 0,9 donne les meilleurs résultats.

Dans notre cas parce que le gain est l'opposé du carré de la distance à l'objectif, on se retrouve avec un γ plus bas que ce qui est communément utilisé. Il aurait fallu que le gain représente l'avancée vers l'objectif : positif si on s'y rapproche et négatif si on s'en éloigne.

3.4.6 Optimisation de la plage de saturation (clip range)

PPO sature la mise à jour de la politique pour éviter une trop grosse mise à jour. Par exemple avec une plage de saturation de $\varepsilon = 0.2$, la différence entre l'ancienne et la nouvelle politique est saturée entre $1 - \varepsilon = 0.8$ et $1 + \varepsilon = 1.2$.

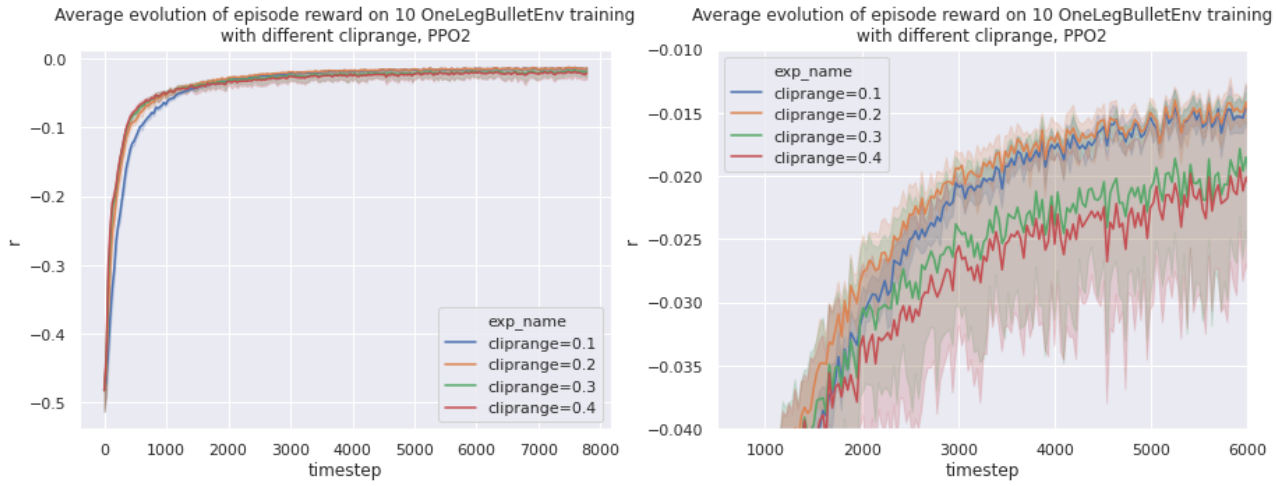

FIG. 17 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement, avec différentes plages de saturation

On observe qu'une plage de 0,1 sature trop la mise à jour de la politique et dégrade lezs performances. Dans l'autre sens, une plage plus grande que 0,2 provoque des mises à jour trop brusques et dégrade les performances et la répétabilité.

3.4.7 Optimisation du rythme d'apprentissage (learning rate)

Le rythme d'apprentissage (learning rate) est un hyperparamètre important. **Il représente la taille des pas d'optimisation de l'optimiseur Adam (Adaptive Moment Estimation).** Un pas plus grand améliora la vitesse mais la politique ne convergera pas aussi bien qu'avec un pas plus petit [14].

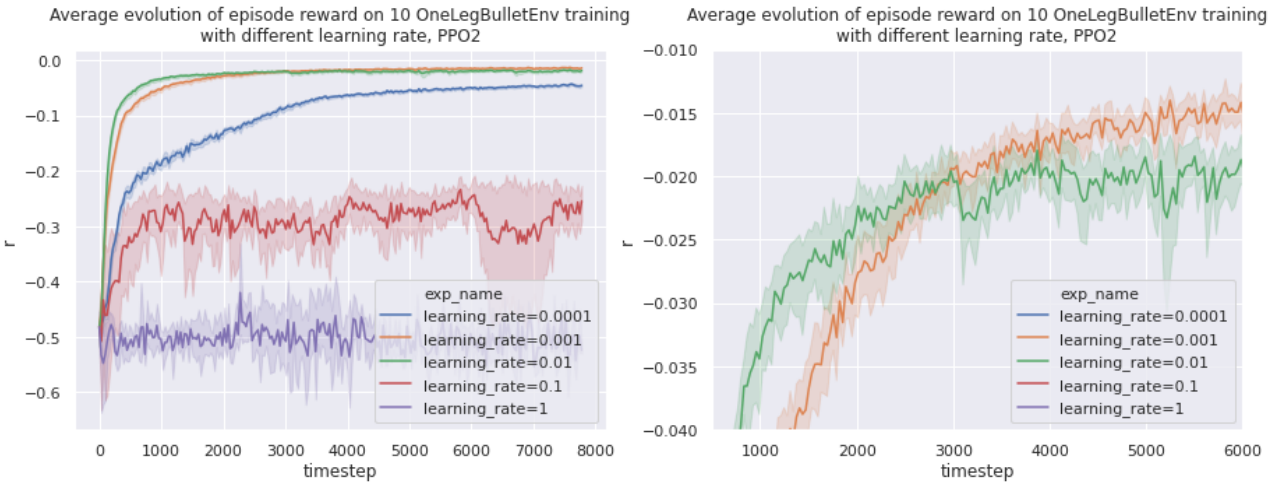

FIG. 18 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement, avec différents rythmes d'apprentissage

Un learning rate de 0.01 donne les meilleurs performances, mais en le diminuant on converge vers une politique légèrement meilleure. Une façon de résoudre cela est de réduire le rythme d'apprentissage dynamiquement au fur et à mesure de l'apprentissage.

3.4.8 Optimisation du facteur de lissage de l'estimateur d'avantage

Le facteur de lissage de l'estimateur d'avantage `lam` permet de définir le lissage λ de l'estimateur de la fonction d'avantage (Generalized Advantage Estimation). Le Generalized Advantage Estimation est une méthode pour estimer la fonction d'avantage qui est utilisée par PPO [15].

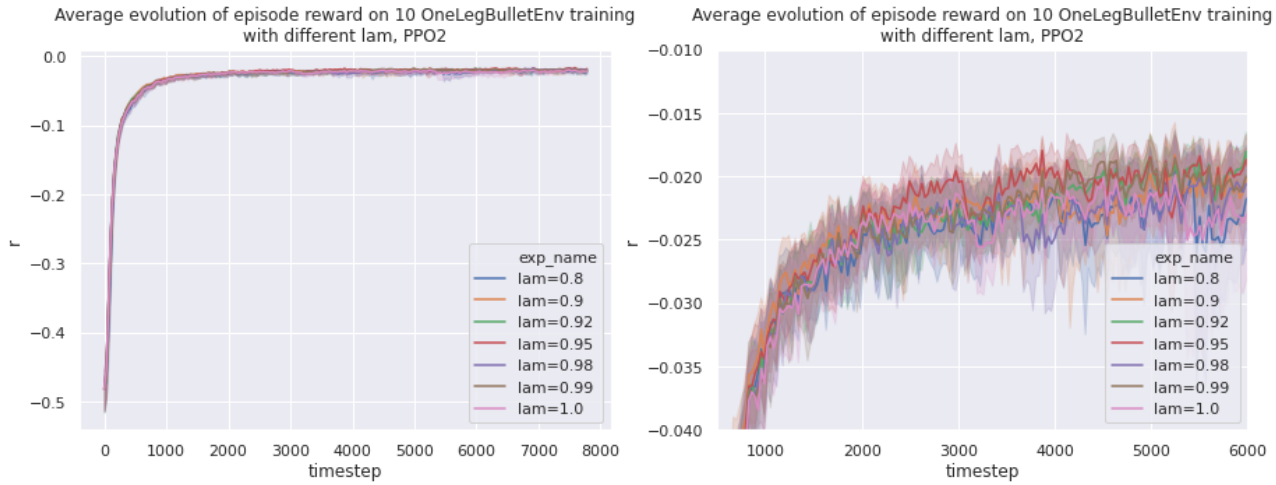

FIG. 19 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement, avec différents facteurs de lissage de l'estimateur d'avantage

Dans notre environnement, ce facteur n'influence pas vraiment l'apprentissage. Même en s'éloignant des valeurs courantes (0.9 à 1.0) les performances restent similaires.

3.4.9 Optimisation du facteur de pondération de l'entropie

`ent_coef` est le coefficient de pondération de l'entropie dans la fonction à optimiser (loss function). Augmenter cette pondération augmentera l'exploration pendant l'entraînement.

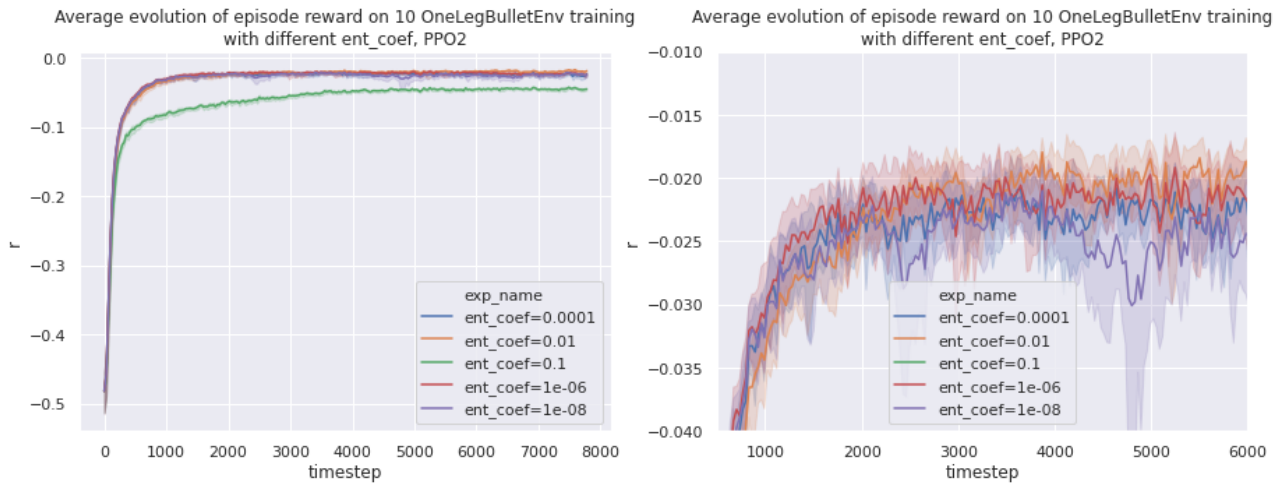

FIG. 20 : Évolution du gain sur un épisode en fonction du nombre de pas de temps effectués lors de l'entraînement, avec différents facteurs de pondération de l'entropie

On observe que `ent_coef=0.01` donne des bons résultats. Augmenter au delà ralentit l'entraînement car on explore trop.

3.5 Transfert de la politique sur le robot réel

Les différents entraînements de la section précédente ont été réalisés par un pas de temps d'environnement de 50 ms. Cela correspond à la période minimale à laquelle on peut observer tous les angles et vitesses des servomoteurs.

Nous avons également construit des environnements simulés qui n'utilisent que des données observables dans la réalité. Les environnements ont ainsi un équivalent « monde réel » qui n'utilise pas PyBullet et qui n'est pas capable de calculer un gain.

Nous n'avons pas testé le transfert du réseau de neurones entraîné sur le robot. Néanmoins nous avons réussi à mettre en place Microconda sur le robot avec les différentes librairies nécessaires.

4 Prise en main de fermes de calculs SLURM

4.1 Création d'une image Docker pour StableBaselines

Réaliser un apprentissage sur nos environnements avec StableBaselines nécessite un certain nombre de dépendances dont TensorFlow1. TensorFlow évolue rapidement avec ses librairies. Cela rend impossible d'installer une ancienne version avec le gestionnaire de paquet d'une distribution Linux ou Anaconda récente sans casser l'arbre de dépendance du système.

Docker est un outil de conteneurisation. Il utilise le système d'isolement du noyau Linux afin de créer un système isolé avec ses propres versions de logiciels et librairies. Il est devenu un outil populaire dans le domaine de l'apprentissage machine car des sociétés comme Nvidia fournissent des images réutilisables optimisées pour ce domaine.

Dans notre cas, une solution est de se créer un environnement conteneurisé avec Docker partant de l'image officielle de TensorFlow avec le support de Nvidia Cuda et Python3. Nous écrivons donc les instructions suivante dans un `Dockerfile`.

```
FROM tensorflow/tensorflow :1.15.2-gpu-py3
RUN apt-get update && \
    apt-get install -y libsm6 libxext6 libxrender-dev cmake zlib1g-dev
       ffmpeg libglvnd0 libgl1 libglx0 libegl1 mesa-utils
RUN pip install --no-cache-dir gym stable-baselines imageio jupyter
   jupyterlab matplotlib optuna seaborn
WORKDIR /tf/kraby
COPY . /tf/kraby
RUN pip install --no-cache-dir -e .

EXPOSE 8888
CMD jupyter lab --ip 0.0.0.0 --no-browser --notebook-dir=/tf/kraby
```

Ainsi nous obtenons l'équivalent d'une machine virtuelle, mais sans émulation d'un noyau Linux, qui contient l'ensemble des logiciels dont nous avons besoin. À son lancement un Jupyter Notebook sera lancé sur le port 8888 de notre machine.

Nous avons également publié cette image. Il suffit alors de lancer l'image `erdnaxe/kraby` pour obtenir cet environnement sur n'importe quelle machine équipée de Docker.

Il se trouve que TensorFlow 1.16 n'est pas installable dans un environnement Anaconda à cause des dépendances pré-installées, mais il est possible de l'installer dans un Miniconda.

4.2 Découverte et prise en main du LabIA

Le LabIA de l'Université Parie-Saclay est une ferme de calculs orchestrée par l'outil SLURM. Un nœud de calcul est composé d'un ou deux GPU et d'une trentaine de CPU en moyenne.

L'idée de SLURM est de décrire des tâches à effectuer dans un fichier script Bash en explicitant les contraintes matérielles en commentaire. Ensuite on envoie ce fichier dans une file d'attente et la tâche sera lancée quand les ressources seront disponibles.

Point important, **une tâche est préemptive**. Elle peut donc se faire tuer à tout moment et c'est à l'utilisateur d'implémenter de quoi sauvegarder son état et de la restaurer.

Contrairement à ce que l'on pensait initialement, le LabIA ne propose pas à ses usagers de solution de conteneurisation tel que Docker ou Singularity. On a donc dû reconstruire un environnement Miniconda en personnalisant les dépendances pour obtenir quelque chose de similaire à notre image Docker.

Après quelques tests, il s'avère que le LabIA est mal dimensionné pour notre apprentissage. Nous avons besoin de beaucoup de ressources CPU pour pouvoir utiliser correctement un GPU Nvidia V100. Les CPU du Lab-IA sont optimisés pour avoir beaucoup de cœurs, mais un cœur n'a qu'une vitesse de 2.2GHz. De plus, nous avons observé des transferts de données très lent entre le disque réseau et la mémoire vive, ce qui rendait la création des environnement beaucoup plus long.

Pour un entraînement avec 32 simulations parallèles, nous avons observé qu'un PC portable avec un i7-8750H et une Nvidia GTX1060 mettait environ 1 minute pour réaliser 250 000 pas de temps d'apprentissage, alors qu'un nœud du LabIA avec 32 CPU et une Nvidia V100 mettait 4 à 5 minutes. La majorité du temps perdu sur le nœud était lors de l'initialisation des environnements.

4.3 Découverte et prise en main de Jean Zay

Quelques semaines après avoir obtenu l'accès au LabIA, nous avons obtenu un accès préparatoire à Jean Zay. Nous avons alors testé si des problèmes similaires de performance apparaissaient.

Jean-Zay est une ferme de calculs plus récente. Singularity est installé et permet aux utilisateurs d'utiliser des images Docker pour leurs tâches SLURM. Néanmoins nous n'avions pas obtenu les droits nécessaire pour son utilisation.

Nous effectuons les mêmes entraînements que sur le LabIA et cette fois nous obtenons des performances très similaires entre le PC portable et un nœud de Jean Zay. Néanmoins nous avions demandé des nœuds GPU et non CPU, ce qui nous empêchait de démarrer des tâches avec plus de CPU et moins de GPU. Il devrait donc être possible de réaliser les apprentissages encore plus vite sur un unique nœud de Jean Zay.

Pour vraiment profiter de la puissance de ces fermes de calculs il aurait fallu utiliser une librairies comme RLlib étant capable de séparer l'apprentissage sur des nœuds GPU et la génération des données sur des nœuds CPU.

5 Conclusion

Ce stage a été l'occasion de découvrir et explorer de nombreux domaines et outils. Non seulement nous avons pu mettre en place une simulation physique fonctionnelle du robot, mais nous avons appliqué des algorithmes proche de l'état de l'art en renforcement profond dessus.

Réaliser l'apprentissage de la locomotion d'un robot est une tâche qui nécessite de nombreuses connaissances à la fois en électronique, mécanique mais également en apprentissage machine et en apprentissage par renforcement. Nous avons bloqué de nombreuses heures sur des problématiques communes à un projet d'apprentissage de locomotion et nous avons documenté au fur et à mesure les solutions.

Tout au long de ce stage, nous avons documenté la partie technique et la mise en place des environnements de simulation. En utilisant cette documentation une personne peut construire une simulation et réaliser l'apprentissage de la locomotion sur un autre robot.

Vers la fin de ce stage, nous avons profité d'avoir un environnement relativement simple pour pouvoir mettre en avant l'effet des différents hyperparamètres liés à l'apprentissage. Nous avons créé ainsi une ressource qui pourra être utile à des futurs débutant dans ce domaine pour comprendre et prendre en main l'optimisation des hyperparamètres.

Il serait envisageable de continuer le projet en optimisant les hyperparamètres pour pouvoir apprendre sur l'ensemble du robot, puis tester des idées tel que pénaliser la consommation énergétique du robot dans la fonction de gain. Des nouvelles avancées dans le domaine de l'apprentissage profond pourrait également être testées tel qu'amorcer l'apprentissage en imitant un mouvement existant.

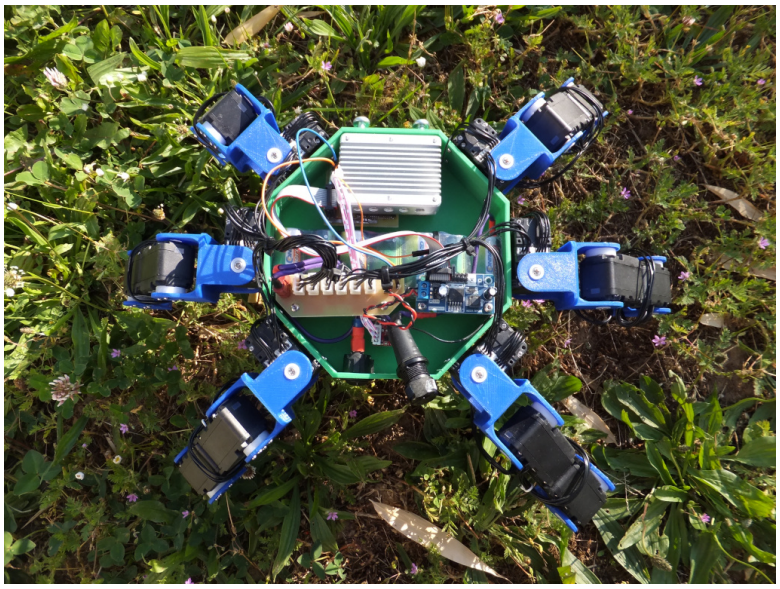

Fig. 21 : Robot hexapode vu de dessus

Nous tenons à remercier Anthony Juton et Justin Carpentier pour les précieux conseils et encadrement apporté au cours de ce stage.

Références

[1] F. Grimminger, A. Meduri, M. Khadiv, J. Viereck, M. Wüthrich, M. Naveau, V. Berenz, S. Heim, F. Widmaier, T. Flayols, J. Fiene, A. Badri-Spröwitz et L. Righetti, "An Open Torque-Controlled Modular Robot Architecture for Legged Locomotion Research," *IEEE Robotics and Automation Letters*, t. 5, n° 2, p. 3650-3657, 2020. doi : 10.1109/LRA.2020.2976639.

[2] M. Hennerich, *Linux Industrial I/O Subsystem*, https://wiki.analog.com/software/linux/docs/iio/iio, 2012.

[3] S. Madgwick, R. Vaidyanathan et A. Harrison, "An Efficient Orientation Filter for Inertial Measurement Units (IMUs) and Magnetic Angular Rate and Gravity (MARG) Sensor Arrays," Department of Mechanical Engineering, rapp. tech., avr. 2010. adresse : http://www.scribd.com/doc/29754518/A-Efficient-Orientation-Filter-for-IMUs-and-MARG-Sensor-Arrays.

[4] E. Coumans et Y. Bai, *PyBullet, a Python module for physics simulation for games, robotics and machine learning*, 2016. adresse : http://pybullet.org.

[5] 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," 2018. adresse : https://arxiv.org/abs/1804.10332.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang et W. Zaremba, *OpenAI Gym*, 2016. adresse : https://arxiv.org/abs/1606.01540.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford et O. Klimov, "Proximal Policy Optimization Algorithms.," *CoRR*, 2017. adresse : https://arxiv.org/abs/1707.06347.

[8] Y. Duan, X. Chen, R. Houthooft, J. Schulman et P. Abbeel, "Benchmarking Deep Reinforcement Learning for Continuous Control.," *CoRR*, 2016. adresse : https://arxiv.org/abs/1604.06778.

[9] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018. adresse : https://spinningup.openai.com.

[10] 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*, 2018. adresse : https://github.com/hill-a/stable-baselines.

[11] OpenAI, *OpenAI Five*, https://blog.openai.com/openai-five/, 2018.

[12] A. Raffin, *RL Baselines Zoo*, 2018. adresse : https://github.com/araffin/rl-baselines-zoo.

[13] T. Akiba, S. Sano, T. Yanase, T. Ohta et M. Koyama, "Optuna : A Next-generation Hyperparameter Optimization Framework.," *CoRR*, 2019. adresse : http://arxiv.org/abs/1907.10902.

[14] D. P. Kingma et J. Ba, *Adam : A Method for Stochastic Optimization*, 2014. adresse : https://arxiv.org/abs/1412.6980.

[15] J. Schulman, P. Moritz, S. Levine, M. Jordan et P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2015. adresse : https://arxiv.org/abs/1506.02438.