IFT6163 Robot Learning Project: Learning Legged Locomotion Policies

Simon Chamorro, Lucas Maes, Valliappan CA

April 27, 2022

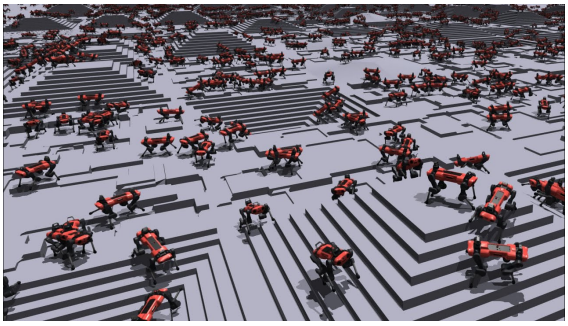
Abstract

Learning an end-to-end locomotion policy for a quadruped robot in the real world is a difficult problem, mainly because of its complexity, the sample inefficiency of our learning algorithms, and the ability to deal with uncertainty. However, Sim2Real methods try to tackle this problem by learning in a simulated environment, where data is easy to generate and where collisions do not cause any real damages. In this report, we aim to make a cheap and flimsy robot, i.e. the Stanford Pupper, learn to walk in the massively parallelizable simulator Isaac Gym using curriculum learning and a hand-crafted reward function and transfer this policy to the real robot. We use PPO to learn our locomotion policy in simulation. Finally, we transfer our learned policy to a real physical robot and qualitatively analyse its performance. Additionally, we also propose an Isaac Gym compatible model of the Stanford Pupper as well as the Unitree Go1. Our work also contains an ablation study of reward terms and a study of different environment parameters to understand the learning process better.

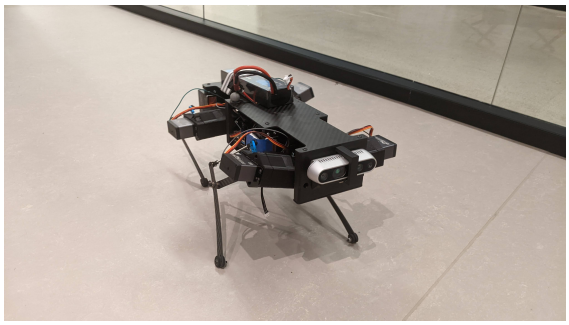
1 [4 pts] Project Introduction

Legged robot locomotion is a very interesting area of research, since legged robots can navigate complex environments which are inaccessible to traditional wheeled robots. Stairs are a good example of a very common obstacle that makes this kind of locomotion more robust. However, developing locomotion policies for legged robots is difficult. Hard-coded, trajectory based techniques have shown to work well, but mostly on flat surfaces [Wettergreen and Thorpe, 1992]. Learning locomotion policies is therefore a good avenue to develop legged robots that are robust and can adapt to different kinds of terrain. This research area has gained a lot of popularity in the past few years [Yue, 2020].

In this work, we use the highly parallelizable GPU-based simulator Isaac Gym (shown in figure 1a) to train locomotion policies for legged robots in simulation. We first study the learning process by experimenting with different environment parameters and doing an ablation study of the different reward terms that are used. Then, we learn a locomotion policy for the Stanford Pupper in simulation and transfer it to the real robot. Figure 1b shows a picture of the robot in action. We analyse the transfer results qualitatively.



(a) Isaac Gym Simulator



(b) Stanford Pupper

Figure 1: Figure 1a is from Rudin et al. [2022] and shows parallelized training of thousands of legged robots. Figure 1b shows the Stanford Pupper, which we will use to transfer learned policies.

2 [2 pts] What is the related work?

In recent years, deep reinforcement learning has been extensively studied in many real-world scenarios, including in the domain of legged robotics locomotion [Hwangbo et al., 2019, Lee et al., 2020]. One major limitation is that data collection may turn out to be very expensive in the real world. That's why researchers prefer to train a policy in a simulated environment, which is safer and less costly for sampling, in order to re-use that learned policy in the real world. The predominant approach to achieve this is named Sim2Real and consists of having one or multiple robots simultaneously collect data for training on a CPU based simulation as proposed by traditional simulators like Mujoco, PyBullet, Rainer and then, extracting the learned policy and deploying it in a real world robot. Although this method has shown some success Tan et al. [2018], it suffers from limitations, such as training time (from multiple days to even weeks). To tackle this problem, an end-to-end high performance GPU accelerated robotics simulation platform was created [Makoviychuk et al., 2021], which allows reducing the training time to a magnitude of minutes while simulating thousands of robots. Another problem when dealing with legged robotics locomotion and Sim2Real is how to learn a policy which will be efficient when transferred from the simulated environment to a real-world robot. One of the most promising recent approaches seems to be to use massive parallel computation power to simulate and train a lot of agents that will upgrade their policy by running a version of PPO (Proximal Policy Optimization) that uses a simplified learning objective in a game-inspired curriculum [Rudin et al., 2022], which leads to impressive results. However, this approach has some limitations, especially in the real-world part, such as dealing with imperfect terrain mapping or state estimation drift. In this work, we start by reproducing the results in [Rudin et al., 2022], but with different robot models.

We adapted the method described in the paper to our robot as well as our available sensors. In the paper, the robot uses different observations: its linear and angular velocities, gravity measurements, joint positions and velocities, as well as ground height measurements. From this data, it learns a locomotion policy that is robust to different terrains. It is able to navigate through stairs, inclined planes, and random objects of up to 0.2 meters. We

perform some ablations to the reward terms and study different environment parameters to understand the learning process better. This study is inspired by [Reda et al., 2020], where the authors argue that environment design plays a huge role in learning, but is often overlooked. Finally, we transfer one of our learned policies to a physical robot, the Stanford Pupper, and do a qualitative analysis.

3 [2 pts] What background framework have you used?

Mathematical background:

Classical RL:

For this project, we consider a standard reinforcement learning setup using an environment E (with discrete time steps) and an agent A. At each time step t, A receives an observation o_t from E, takes an action $a_t \in \mathbb{R}^n$ and gets a scalar reward $r_t \in \mathbb{R}$ (from E). In this project, the task is fully-observable, meaning that $o_t = s_t$ where s_t is the real state of the environment. The environment is modeled as a Markov Decision process which is a tuple $< \mathcal{S}, \mathcal{A}, p_0, \mathcal{T}, \mathcal{R}, \gamma >$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, p_0 is the probability distribution over the initial state s_0, $\mathcal{T}$ is the transition dynamics operator giving $p(s_{t+1}|s_t, a_t)$, $\mathcal{R}$ is a reward function and $\gamma \in [0, 1]$ the discount factor. The behavior of the agent A is called a policy, π which is a mapping $\mathcal{S} \to \mathcal{P}(\mathcal{A})$ from state space $\mathcal{S}$ and probability distribution $\mathcal{P}(.)$ over the action space, $\mathcal{A}$ meaning that the policy is predicting how the agent will act in a particular state. The goal of the policy is to maximize the expected return, which is defined as $J(\theta) = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi_\theta}[\sum_{i=1}^{T} \gamma^{(i-1)} r(s_i, a_i)]$.

Policy Gradient:

Policy gradient methods are a way of solving an MDP by learning a policy step by step through an estimation of the policy gradient used with a gradient ascent algorithm, from which the update rule is given by $\theta_{t+1} = \theta_t + \alpha \hat{g}_t$ where α is the learning rate and $\hat{g}_t$ the estimation of our gradient, typically $\mathbb{E}[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t]$ where $\hat{A}_t$ is estimation of the advantage function at time step t. Policy gradient is what we call a model free method, meaning that it is not using any world model in its learning process. It is also an **on-policy** method, as it makes updates using transitions that were obtained by following the current policy.

For all our experiments, we will use the PPO (Proximal Policy Optimization) [1] , an improved version of vanilla policy gradient, implementation used in the paper. The difference with vanilla PPO is that they implemented the algorithm with a simplified version of the learning objective. Figure 2 shows the pseudocode for the modified algorithm version.

The Vanilla PPO objective is,

[1] Spinning-up PPO: https://spinningup.openai.com/en/latest/algorithms/ppo.html

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s,a), \quad \text{clip}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right)$$

using Spinning-up, the PPO objective becomes

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s,a), \quad g\left(\epsilon, A^{\pi_{\theta_k}}(s,a)\right)\right)$$

where,

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

and ϵ is a hyperparameter typically set at 0.2.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t,a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t,a_t))\right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Figure 2: PPO-Clip algorithm from spinningup.openai.com

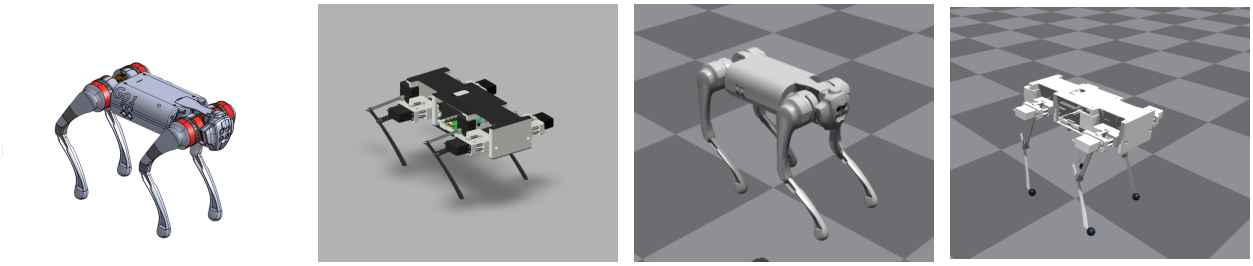
4 [4 pts] Project Method, How will the method work?

In this section, we present our method for learning a locomotion policy in simulation and transferring it to a real quadruped robot. We discuss the robot modeling, the training in simulation, the setup of the physical robot and the transfer of the learned policy.

4.1 Modeling Robots in Simulation

First of all, we need a simulated model of the robots we want to train in order to use them in simulation. Typically, robotics simulators use URDF files to describe the physical parameters

of robots and load them. This is the case of the Isaac Gym simulator. Initially, we hoped to work with the Unitree Go1 robot, but the settled for the Stanford Pupper due to time and shipping constraints.



(a) Go1 CAD (b) Pupper CAD (c) Go1 in IG (d) Pupper in IG

Figure 3: **Robot Models.** This figure shows the Unitree Go1 as well as the Stanford Pupper CAD models as well as their integration in Isaac Gym.

Figure 3. For the Go1 robot, Unitree provides a description file for Gazebo in their ROS packages[2]. This model was adapted to work with Isaac Gym by changing the link structure. However, there was no need to modify the physical parameters as they are already very accurate. As for the Stanford Pupper, an XML description file is available with the official PyBullet simulation. This type of file is not well supported in Isaac Gym, so the dimensions were used to create a URDF file. Additionally, the masses did not exactly match those of the Pupper we used, so they were modified to math our robot better.

4.2 Training in Isaac Gym with PPO

We use the Isaac Gym simulator and the PPO algorithm to train our locomotion policies in simulation.

4.2.1 Isaac Gym: Highly Parallelizable GPU-Based Simulator

Learning a good real-world policy is a difficult task, as RL algorithms are not very sample efficient and real-world data is very expensive to collect. A well studied solution to tackle these issues consists of learning a policy in simulation to later transfer it into a robot in the real-world. We choose to use the Isaac Gym simulator, as it provides trade off between fidelity of the simulation and computational efficiency. Furthermore, Isaac Gym provides a way to make all the computation directly on GPU, avoiding a bottleneck of information signals flooding between GPU and CPU and vice-versa. The well optimized code allows us to train the Go1 policy and the Stanford Pupper policy (based on PPO) to walk on a flat terrain for 500 steps in less than 5 minutes on a `RTX 3080TI`.

4.2.2 PPO Algorithm and Implementation

We use the PPO implementation that the authors of [Rudin et al., 2022] used to learn locomotion policies for the Anymal C, Cassie, and Unitree A1 robots. The code is provided

[2]https://github.com/unitreerobotics/unitree_ros

as a specific library built to support a large batch size on Isaac Gym in order to still be sample and computationally efficient during the training process, as the number of training environments can be large. The library, named `rsl_rl`[3], contains a specific version of PPO (Proximal Policy Optimization) [Schulman et al., 2017] based the open AI's spinning up library.

4.3 Setting up the Physical Pupper

By default, the Stanford Pupper has no proprioceptive sensors to estimate its state and construct an observation for an RL agent. When training policies in Isaac Gym, we use the following information to construct observations:

- Robot linear and angular velocity;
- Joint angles and velocities;
- Linear and angular velocity targets;
- Last action.

While the actions and target are easy to obtain, we need sensors for the robot velocity as well as the joint information. We used an Inertial Measurement Unit to obtain the robot velocity and motor encoders to obtain joint information.

4.3.1 Inertial Measurement Unit for State Estimation

We use an Inertial Measurement Unit (IMU) from the BNO08X family to measure the robot state. It has a triaxial accelerometer, triaxial gyroscope, and magnetometer. The BNO080 IMU produces accurate rotation vector headings, excellently suited for VR and other heading applications, with a static rotation error of two degrees or less. All the sensor data is combined and drift-corrected into meaningful, accurate IMU information. This IMU breakout board has also been equipped with two I2C Qwiic connectors, in order to make interfacing with I2C, which we use to communicate with the Raspberry Pi. We've also written a Kalman filter to avoid the noisy estimation of the acceleration, gyro and magnetometer readings, which we integrate to obtain velocity information. The working of the kalman filter implementation is yet to be tested with the IMU sensor data from the Pupper, but we did test the performance with a predefined sinusoidal input. The performance is shown in Figure 4.

4.3.2 Motor Encoders for Joint Information

The Stanford Pupper uses 12 JX Ecoboost CLS6336HV servo motors as actuators (3 for each leg). These motors are controlled via a PWM signal. An angle target is sent, and an internal driver moves them to the desired position. Therefore, they have an integrated encoder to know what the motor position is. However, this signal is not communicated outside the motor. To obtain the joint states, we soldered a wire to read the encoder signal for each

[3] https://github.com/leggedrobotics/rsl_rl

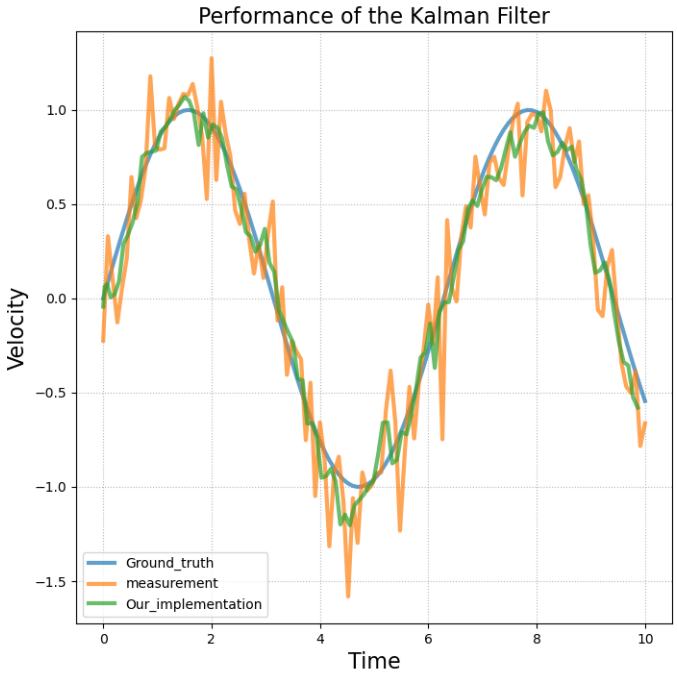
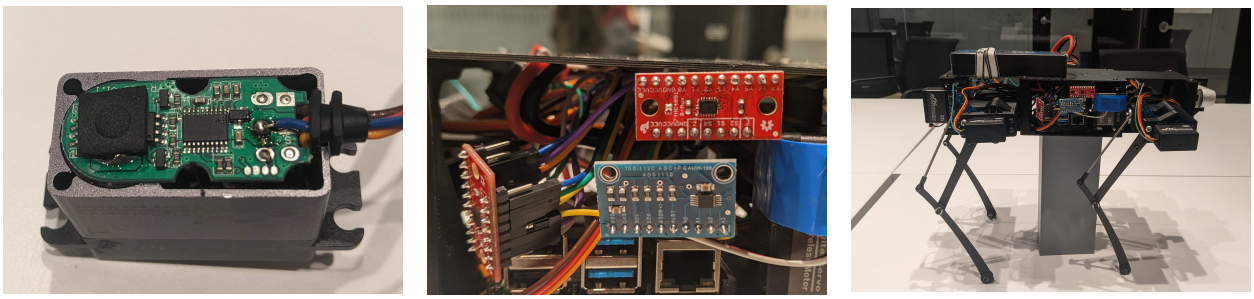


Figure 4: **Kalman filter Performance on Simulated Data**.

motor. Then, we wired these wires to two 8-channel multiplexers, which then communicate with an analog-to-digital converted. The chips are controlled via GPIO pins from the Pupper Raspberry Pi using CircuitPython[4]. Figure 5 shows the process we went through to add the encoders. After the encoders and chips were wired, we wrote code to do a calibration routine and then read the encoder values at run time.



(a) Servo Motor (b) Multiplexers and Converter (c) Pupper Calibration

Figure 5: **Adding Joint Encoders.** This figure illustrates how the internal motor encoders are used to extract the joint state.

4.4 Transferring the learned policy

Once the physical Pupper is equipped with all the necessary sensors to provide an observation and all the actuators to execute actions, we are ready to transfer our learned policy from

[4]https://github.com/adafruit/circuitpython

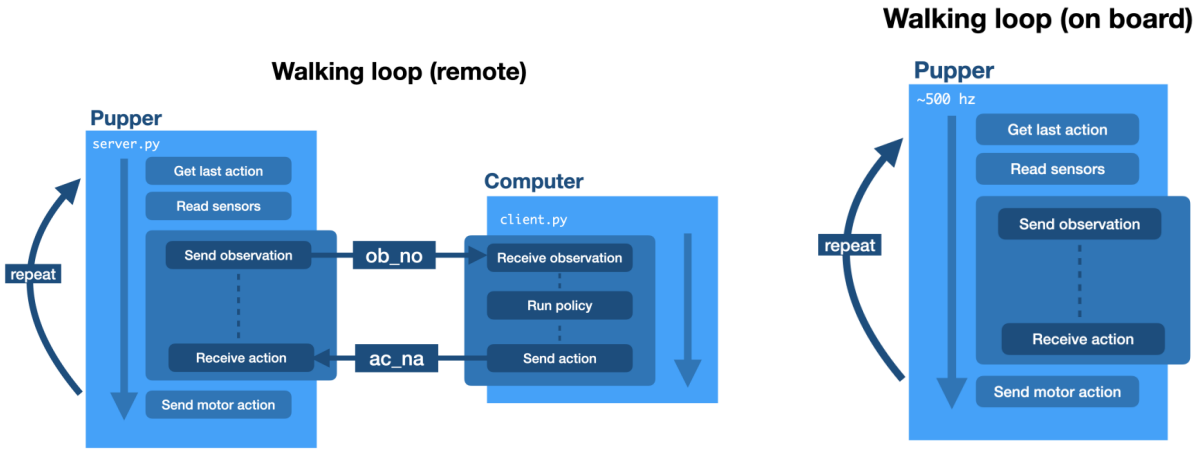
simulation. However, we first validate our policy in the PyBullet Pupper simulation provided in the official Stanford Pupper release.

4.4.1 Validation in the PyBullet Simulator

Before transferring the learned policy directly to the real world pupper, we found it useful to test the policy in a highly accurate simulation of the Pupper in PyBullet which exactly works as the Pupper and runs the same software that runs on-board. This validations allows us to be sure the output of the policy network is sending correct commands to the right joints and overall has a reasonable behavior to avoid possible damage to the robot when testing in the real world. One may wonder why we didn't directly train the locomotion policy on the PyBullet simulation? In fact, Isaac Gym is much more efficient in terms of computation and parallelization. Being completely GPU-based, we are able to simulate 4096 robots at once and learn a locomotion policy on flat terrain in under 5 minutes on a single machine.

4.4.2 Real Robot

There are two ways to run the policy on the real robot: 1) streaming the observation from the on-board computer to an external machine that runs the policy. We then send back the command to the Pupper via streaming as well and 2) computing the action from the observations by running the PyTorch model on-board. The first approach being remote and the second one is on-board. In the remote approach, the Pupper (Raspberry-pi) acts as the server, creates observations from the last action and sensor data, sends it to the external machine which acts as a client. The client computes the actions corresponding to the observation using the learned policy and sends it back to the server. In the case of on-board approach, the observation to action computation is completely done on Raspberry pi. The work flow is shown in figures 6a and 6b corresponding to both the approaches respectively. We wanted to compare both approaches to see which one would be more suitable. Table 1 presents the latency of the responses in both scenarios. We can see that the external computer does compute the actions much faster than the on-board, but the time consumed for streaming causes high delay. The response time is $\sim 5x$ slower in the case of remote compared to the on-board computation since the model used for the policy is very light. Hence for this work we prefer on-board computation.



(a) Remote Approach.

(b) On-board Approach.

Figure 6: **Real Robot workflow illustration**

Approach	Policy Inference Time	Network Delay	Total Loop Time
Remote	1.14ms	10.59ms	11.73ms (85.221 Hz)
On-board	1.83ms	0ms	1.83ms (545Hz)

Table 1: Shows the difference between running the policy on-board and on an external computer.

5 [2 pts] What new skills are you learning from this project?

1. This was our first hands-on experience with Isaac gym, High performance GPU based physics simulation for robot learning.

2. We had hands-on experience working with the Stanford Pupper. We had to create the URDF file to simulate the robot. This was a great learning experience, where we had to understand the physics of the Pupper robot in order to make sure that the behaviour of the robot is natural in the Isaac Gym environment.

3. Our work extends from the simulated environment to the real world and gives us a chance to visualize and analyze the performance gap between the on-board sensor data and perfect data. This is a valuable experience for real-world robotic learning deployment.

4. We learned that it was difficult to train a policy on a real robot, especially when the data from sensors is very noisy. Even if the policy was not successfully transferred,

our experiments made us learn much more about robot learning than if we had just used simulation. First, because we had to add components to the robot such as new sensors. Second, when we tried to debug things, that forced us to have an in-depth understanding of how all the components of the robot work and how to efficiently control it in order to come up with creative ideas to try to debug it.

5. We had the opportunity to gain experience in team working. Each member of the group collaborated with other members from different backgrounds. Each team member has expertise in different areas such as robotics, Deep Learning, and Reinforcement learning, through this project we were able to share our knowledge.

6 [6 pts] Experiments

In this section we present the results for our project. First, we study the locomotion task by doing preliminary experiments in the Isaac Gym simulator. Then, we want to validate that our robot models are good and that we are able to learn locomotion policies that match the performance of other robots reported in [Lee et al., 2020]. Finally, we test our learned policy, first in the Pybullet simulator, then on the real Stanford Pupper.

6.1 Understanding the Locomotion Task

Our project aims to learn a locomotion policy for a quadruped on flat terrain in the real-world. However, we found it useful to study the locomotion task first. The agent goal is to go as far as possible from its origin point, following a direction and an angle determined randomly at the beginning of an episode. We make an ablation study over the nine rewards terms used in [Lee et al., 2020]. We also experiment with different environments parameters. This helps us have a better understanding of what affect the training process and may result in better results in the real-world. Studying rewards and environment design choices are not a new thing and have already been studied in [Reda et al., 2020]. More details about our environment study can be found in this wandb report: Legged Locomotion Environment Experiments in Isaac Gym.

6.1.1 Reward Terms Ablation Study

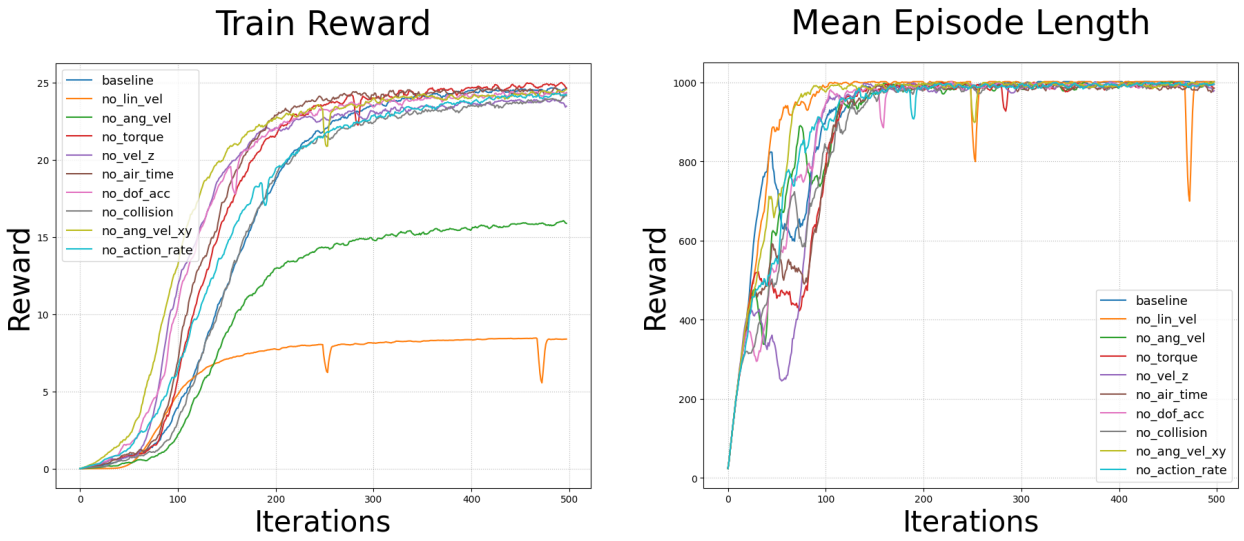
We perform an ablation study on the reward terms that are used to train our locomotion policies in Isaac Gym. Here are the different terms that are used to compute the reward:

- **lin_vel**: Positive reward for tracking target linear velocity.
- **ang_vel**: Positive reward for tracking target angular velocity.
- **torque**: Penalty proportional to torque usage.
- **vel_z**: Penalizes linear velocity along the vertical axis.
- **air_time**: Rewards feet air time.

- **dof_acc**: Penalizes joint acceleration.
- **collision**: Penalizes collisions.
- **ang_vel_xy**: Penalizes angular velocity on the XY plane.
- **action_rate**: Penalty proportional to the action variation from the last time step.

In figure 7, we present an ablation study of the reward terms used to train the Go1 on flat terrain. As shown in 7a, the most important reward terms seem to be the linear and the angular velocities, mainly because they are the principal components to have a moving agent reaching the goal. Other reward terms reach more or less the same results as the baseline. This can be explained by the fact that they are mainly penalty terms so removing them should make the agent gain reward but learn a worse locomotion policy. For the mean episode length (7b), the runs that are not using linear and angular velocities achieve longer episode lengths, but this is due to a static agent. Additionally, removing the penalty for the velocity on the Z axis leads to the worst mean episode length. This might be due to a more unstable robot in the simulation increasing the likelihood for the robot to fall and terminate the episode with a collision. More figures of the reward ablation study are available as supplementary material on our wandb report linked in section 6.1.

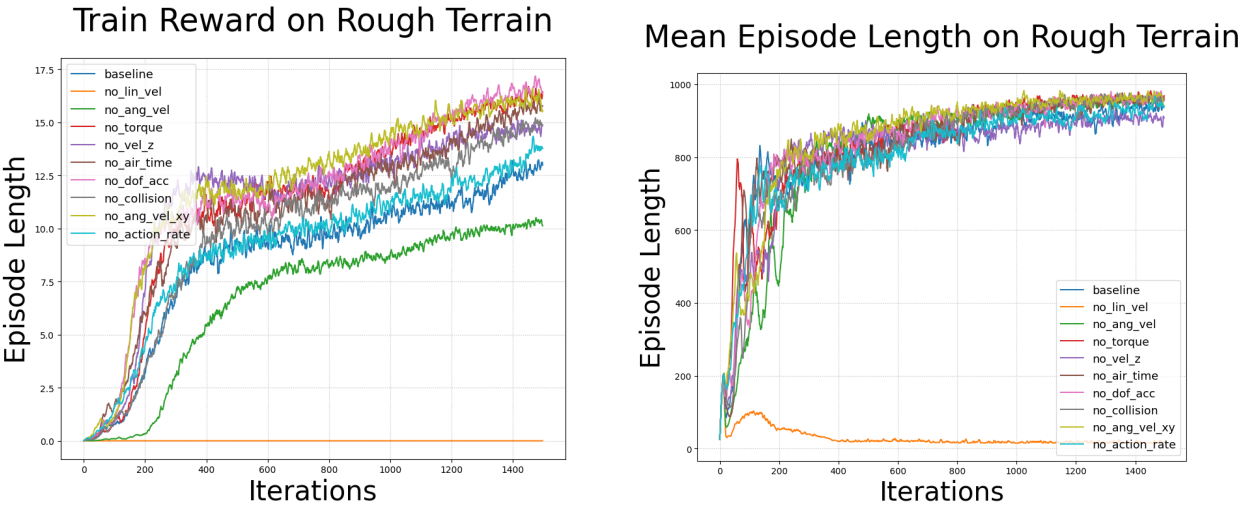
Figure 8 shows our ablation study of the reward terms used to train the Go1 on rough terrain using a terrain curriculum. As we expected, the training reward shown in 8a is generally inferior to the values on flat terrain. It is interesting that the dispersal over runs is greater on rough terrain than it was on flat terrain. This might be sue to a more complex environment to solve. Therefore, we can better see what penalty is more important to learn a good policy. Similarly to flat terrain, the obvious important terms to converge are the linear and angular velocities. Another penalty that seems important on rough terrain is the action rate, which is a penalty proportional to the action variation from the last time step, we interpreted that result as the need for the robot to conserve its trajectory and not deviate suddenly to make fluid movements. For the mean episode length (8b), besides removing linear velocity, removing a particular reward term doesn't seem to change much the length of the episodes.



(a) Reward (b) Mean Episode Length

Figure 7: **Reward ablation analysis on Isaac Gym flat terrain**

It would be interesting to explore different reward terms, as this might not be the optimal configuration. Every term was added with a specific intuition, for example, the air time term encourages the robot to learn a gait early on in training. It would be interesting to explore different reward terms. We leave these ideas for future work.



(a) Reward (b) Mean Episode Length

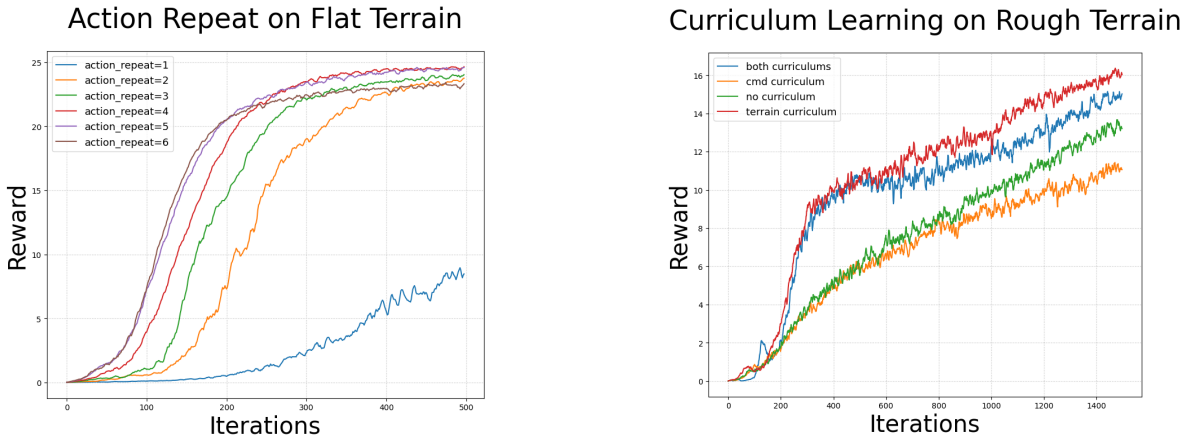
Figure 8: **Reward ablation analysis on Isaac Gym rough terrain using curriculum**

6.1.2 Locomotion Environment Parameters Study

We studied multiple environment parameters and their effect on learning a locomotion policy. Here, we present some of our results on action repeat, torque limits, and curriculum learning. See our wandb report linked in section 6.1 for our complete study.

Action Repeat: This parameter is the number of simulator steps for which the same policy action is repeated. We notice that action repeat helps accelerate training up to a certain point. At an action repeat of 6, the reward starts to slightly drop and the metric of correct behavior shows a small drop in performance. The optimal value is 4 or 5 for a good trade off between skipping frames for learning efficiency and having a policy that is reactive enough. Results are shown in figure 9a.

Curriculum Learning: The command curriculum boolean parameter determines whether the velocity targets for the robot are increasingly hard or random from the beginning. The terrain curriculum boolean parameter determines whether the terrain is increasingly hard or random from the beginning. We notice that the terrain curriculum really helps accelerate the training process. On the other hand, a command curriculum is not useful and even degrades performance. Results are shown in figure 9b.

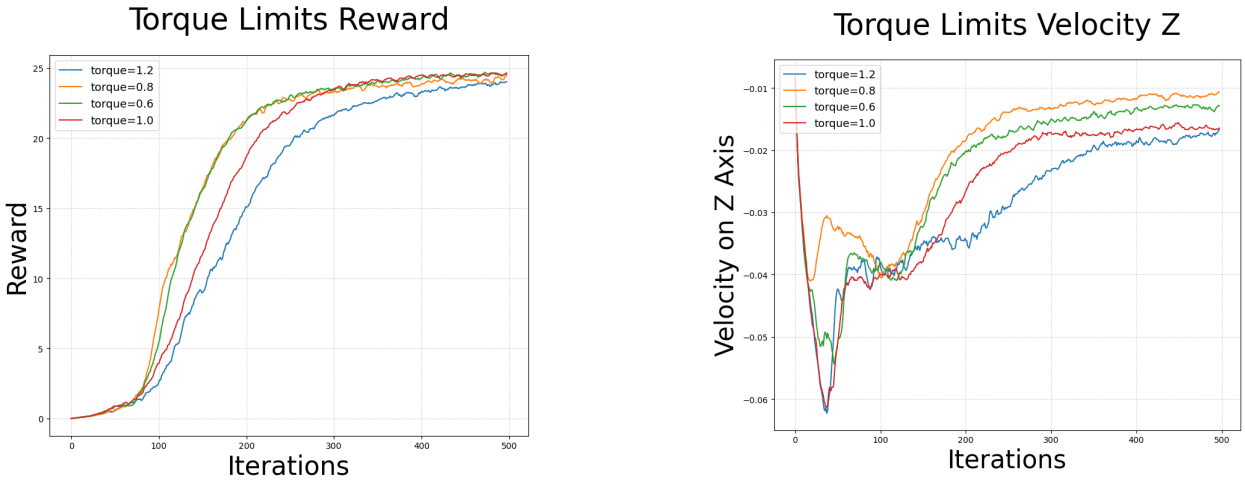


(a) Action Repeat

(b) Curriculum Learning

Figure 9: **Action Repeat and Curriculum Learning Experiments.**

Torque Limits: We modify the actual torque limits of the motors to see the effect of this change on the learned policy. We notice that higher torque limits yield better performance in terms of tracking the desired velocity target. However, the penalties for torque utilization are larger. This is not desirable because it will cause more aggressive behavior. We can see this with the linear velocity in the vertical (z) axis, which is undesirable. This movement in the vertical axis makes the base more unstable. Additionally, it's best to train with the most realistic robot torque for better transfer performance. Results are shown in figure 10.



(a) Reward

(b) Vertical Velocity

Figure 10: **Torque Limits Experiments.**

6.2 Validating our Robot Models

Here, we compare the performance of our Go1 and Pupper to the Unitree A1, which was implemented by the authors from [Lee et al., 2020]. Figure 11 shows our results. We can see that our Go1 model learns a good policy, even slightly better that the A1. On the other hand, the Pupper does not learn at all (in green). We had trouble learning a policy because of the very low mass of the robot, especially at the legs. We tried different strategies such as increasing the mass in simulation and doing a hyper-parameter search. A model of the Pupper with the Go1 mass is our experiment that performed best (in red). This should affect transfer performance, but we think it is acceptable, since the policy outputs position commands to the joints, which are then executed by the robot controllers. Since we kept the robot geometry constant, we think it is acceptable to change its mass to facilitate training.

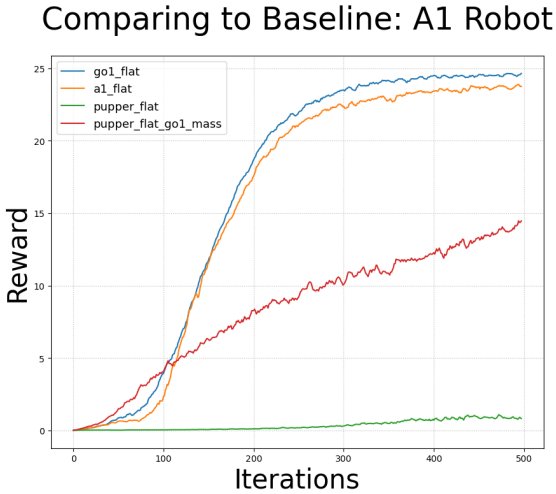
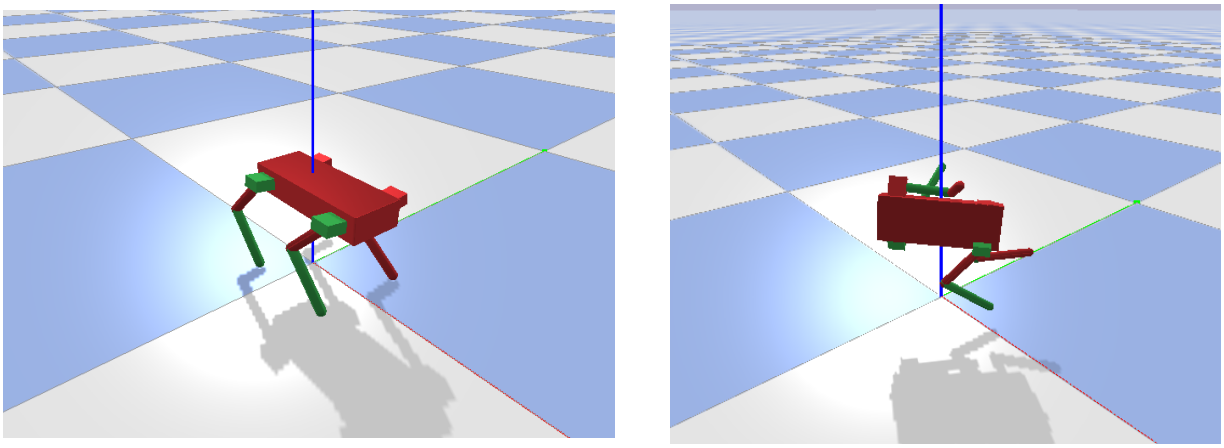


Figure 11: **Comparing Performance to the A1 robot in Isaac Gym on flat terrain.**

6.3 Transferring a Learned Policy

In this section, we present the results for transfer experiments. First, we validate the performance on the PyBullet simulation of the Stanford Pupper. Figure 12 shows the robot model in PyBullet as well as the robot failing to walk. In fact, we notice that the torques applied by the robot seem to be very higg and cause unrealistic behavior in the simulator. This has been reported several times in pybullet [5] In appendix **??**, we provide a comparison of the observation distributions for the Go1 in Isaac Gym, the Pupper in Isaac Gym and the Pupper in PyBullet to better understand this failure to transfer the policy.



(a) Pupper in PyBullet

(b) Fails to Walk

Figure 12: **Validation in PyBullet.**

Finally, we aimed to test the trained policy on the real robot. However, because we were not able to validate its performance on PyBullet, we wanted to first visualize the observation space and make sure it roughly matches the distribution of the observation in Isaac Gym. We found that the IMU sensor is unsufficient for velocity estimation, even when coupled with a Kalman Filter. In fact, all the papers we studied use some method that uses exteroception to get a better velocity estimate and avoid drift, such as GPS, visual odometry, or motion capture systems. Figure 13 shows that our velocity estimated from only IMU is unusable. The ground truth in this figure is very approximate, since we moved the robot ourselves a certain distance to validate the estimation performance. However, this experiment is sufficient to show how much the velocity drifts due to the integration error. We presume that this would be even worse with the jerky movements of the robot. Given more time, we would have opted for a motion capture system to provide the velocity part of the observation to the robot and test our policy on it.

[5] Issue of unrealistic flying behavior (forum link), and Video link.

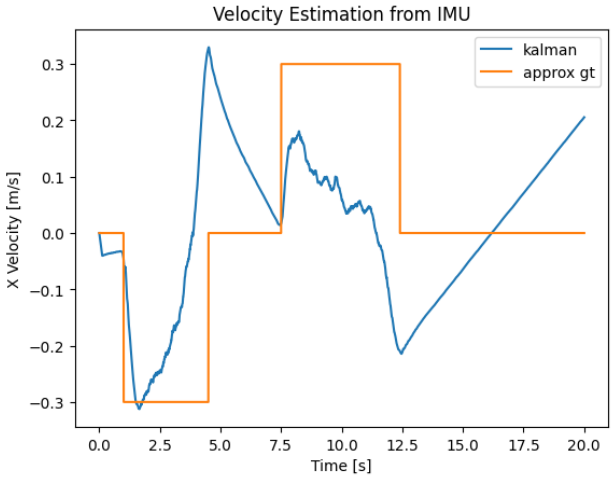


Figure 13: **Velocity Estimation with IMU.**

7 [2 pts] Video Results

A video of our results is available here: https://www.youtube.com/watch?v=RQ0GhKZb5os.

8 [2 pts] Conclusions

This project was the first experience with real robotic hardware for some members of the group. As the Go1 did not arrive in time, we had to switch to the Stanford Puppper, which is very light and sensitive as it is an open source robot and not production-grade quality. This was a challenge that we all appreciated as it reinforced our understanding about working with real hardware and not just in simulation. We had to come up with creative solutions and overcome some problems such as adding sensors to the robot or modifying the robot masses for successful training in simulation. This made us understand the complexity of robot learning in a real world setting and was a very valuable experience. The scope of the project was a bit ambitious at first, and we did not account for delays caused by delivery or by hardware failures, which are inevitable. In the future, we would make sure to scope our projects in a way that we have larger margins for error, especially when working with hardware.

8.1 Acknowledgements

We would like to thank Florian Golemo and Glen Berseth for their valuable suggestions, lending us hardware help to make it works, as well as with the Isaac Gym environment to make the Stanford Pupper URDF file working. We also would like to thank the DIRO technical support that helped us to well setup Isaac Gym on multiple machines to facilitate our experiments.

9 [2 pts] How is the timeline for the project progressing? At least month-by-month granularity, ending with a complete project.

Feburary

1. Have a simulated version of our physical robot in Isaac Gym. **DONE**
2. Train a blind locomotion policy in simulation (only flat terrain and only robot state information). **DONE**
3. Have access to the real robot and familiarize ourselves with it. **DONE**

March

1. Do an ablation study of different reward terms in Isaac Gym. **DONE**
2. Study different environment parameters in Isaac Gym. **DONE**
3. Add physical encoders to the Stanford Pupper to get joint states. **DONE**
4. Setup the Stanford Pupper and make it walk with its hard-coded gait policy. **DONE**

April

1. Add an IMU sensor to the Pupper get velocity information. **DONE**
2. Test our blind policy on the Pupper PyBullet simulator for preliminary validation. **DONE**
3. Develop a system to stream observations from the Pupper to a computer and actions from the computer to the Pupper. **DONE**
4. Test our locomotion policy on the real Stanford Pupper. **NOT DONE:** This could not be done due to the poor velocity estimates for the robot observation and the lack of time to set up the VICON tracking system.
5. Write a complete project report. **DONE**

Note that there have been a few changes to our timeline due to different constraints. First, we were supposed to work with the Unitree Go1, which unfortunately will not be available due to delivery delays. There were also delays with getting access to the physical Stanford Pupper, but we fortunately found a plan B when Florian Golemo from Mila offered to let us use his pupper for this project.

10 [2 pts] How is the work divided?

We managed to make every member of the group work on every step of the project to maximize the learning gain for all as well as provide experience to the others when someone was good in a particular domain. Every member of the group participated to doing initial PPO experiments on Isaac Gym to get acquainted with the algorithm and the simulator. We also had multiple working sessions in person with the real-robot to which all the team members participated. Finally, every member of the team contributed to the drafting of the report.

Simon Chamorro: I modeled the Go1 URDF file and worked on the Pupper URDF together with Lucas. Having some experience with robots, I also lead the work on the physical Pupper, adding the motor encoders, the IMU and writing the code for calibrating and reading the encoders. I worked on the IMU drivers together with Valliappan. Finally, I ran and analysed the experiments on environment parameters in Isaac Gym.

Lucas Maes: I participated in the creation of the URDF for Stanford Pupper and debugged the Pupper simulation in Isaac Gym, trying to scale the agent, to transfer the weight of Go1 into the Pupper or change the URDF file. I also integrated our policy into the PyBullet simulator to validate its performance. The command streaming system between the policy and the Pupper was also made by myself. Finally, I made our video and ran the reward ablation experiments as well as analyzed them.

Valliappan, CA: I participated to debugging the Pupper simulation in Isaac Gym, transfer of policy from Go1 to Pupper, along with Lucas. I also debugged electrical problems when integrating the IMU to the Pupper. I wrote the code for the IMU data and implemented the Kalman filter to smoothen the data. Currently replicating the step() function that we have in Isaac gym to PyBullet, along with Lucas.

Note that Florian Golemo provided a lot of help for setting up the robot and integration the hardware.

References

Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science robotics*, 5(47), 2020.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Daniele Reda, Tianxin Tao, and Michiel van de Panne. Learning to locomote: Understanding how environment design matters for deep reinforcement learning. In *Motion, Interaction and Games*, pages 1–10. 2020.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

David Wettergreen and Chuck Thorpe. Gait generation for legged robots. In *IEEE International Conference on Intelligent Robots and Systems*, 1992.

Jinghong Yue. Learning locomotion for legged robots based on reinforcement learning: A survey. In *2020 International Conference on Electrical Engineering and Control Technologies (CEECT)*, pages 1–7. IEEE, 2020.

Appendix

A Observation Distributions Comparison

In this section, we present a sample of observations for the Go1 in Isaac Gym as well as the Stanford Pupper both in Isaac Gym and in PyBullet to better understand why transfer did not work from a trained policy in Isaac Gym to the PyBullet simulator.

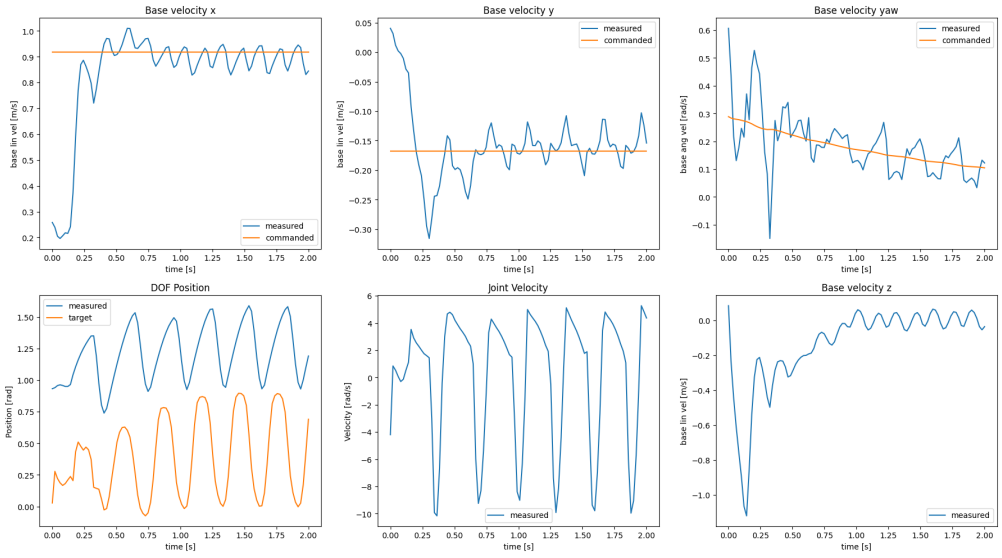


Figure 14: **Go1 Observation Example in Isaac Gym.**

By analysing these plots from figures 14 to 16, the main difference that we think influences the inability to transfer is the joint velocity. In fact, if we look at the joint velocity for the Go1 versus the Pupper, we notice that the Pupper has a higher joint velocity and it has a higher frequency as well. These rapid velocity changes apply high torques in PyBullet, which cause unrealistic behavior, such as the Pupper flying into the air.

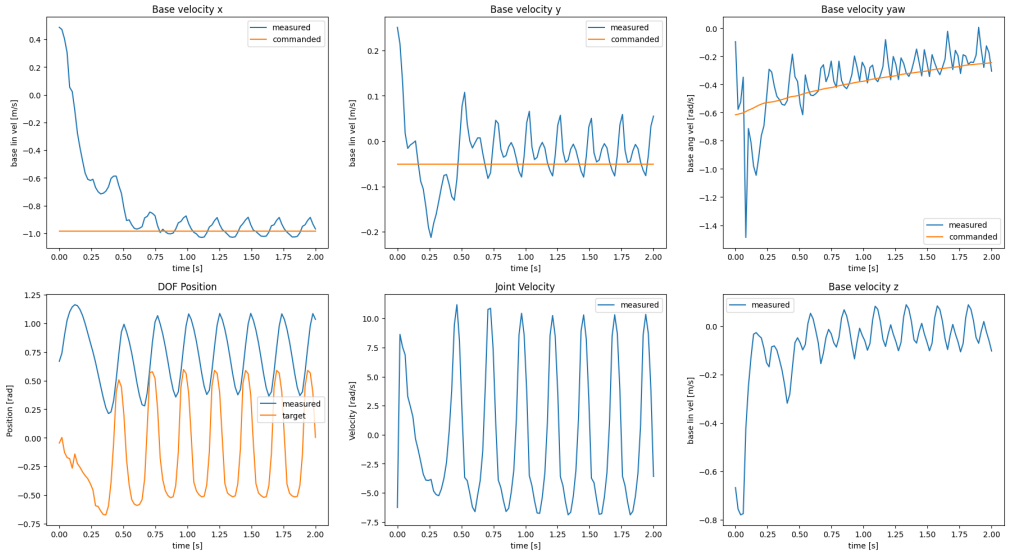


Figure 15: **Pupper Observation Example in Isaac Gym.**

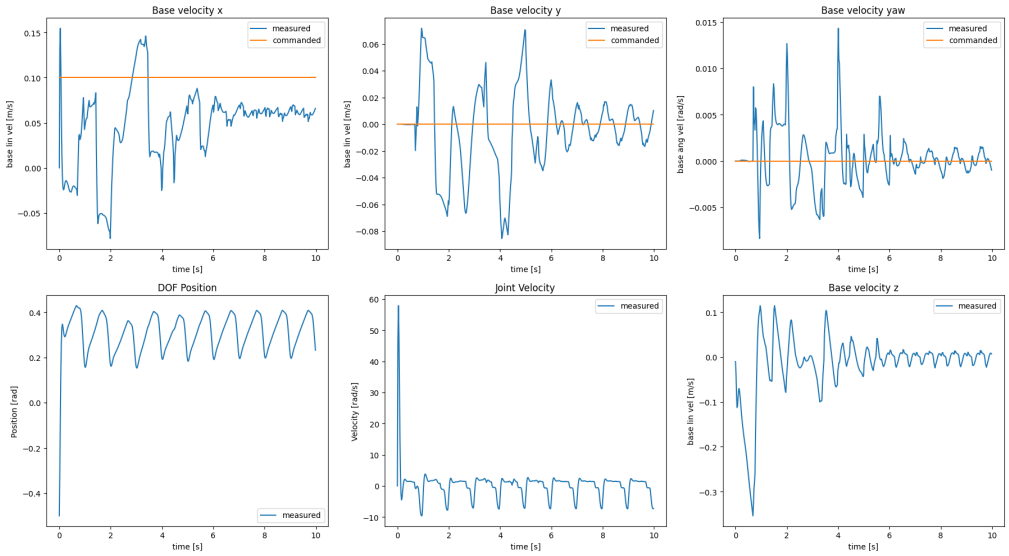


Figure 16: **Pupper Observation Example in PyBullet.**