Deep Reinforcement Learning to Control a Gyroscope

Semester Project

Matthieu Le Cauchois

Professor: Alireza Karimi
Supervisor: Paul-Arthur Dreyfus
Automatic Control Laboratory (LA)

Spring 2020 — Ecole Polytechnique Fédérale de Lausanne

Abstract

Recent breakthroughs in deep reinforcement learning have shown great promise in mastering challenging problems never before tackled by computer algorithms, such as the game of Go. These algorithms have also succeeded in learning control strategies for non-linear dynamical systems without the need for high-fidelity modeling, making the field very attractive for robotics. This recent rise has been enabled by the high representation power of deep neural networks. To leverage this attribute and to compete in the task of continuous control against systems designed using the well-established control theory, careful design of the environment and of the agent must be performed. This work studies this design process applied to the control of a gyroscope. By drawing parallels with control theory, the observation returned by the environment could be wrongfully chosen to be the same as the classical state vector from the state-space representation, leading to non-stationary behaviors. It is also found that using a quadratic reward function similar to LQR can lead to large steady-state errors. The normalized error reward function corrects this behavior significantly by encouraging exploration and thanks to a better scaling of the penalties associated with the initial and steady-state errors. Hyperparameter search shows that deeper networks are essential to allow for more abstract learned representations, which are studied with dimensionality reduction algorithms. Randomization of the dynamics of the simulation is performed, which helps the agent learn more robust behaviors.

Contents

List of Figures

List of Tables

Acronyms

RL Reinforcement Learning

AI Artificial Intelligence

DL Deep Learning

DNN Deep Neural Network

DRL Deep Reinforcement Learning

SOTA State-Of-The-Art

FL Feedback Linearization

MIMO Multi-Input Multi-Output

MDP Markov Decision Process

POMDP Partially Observable Markov Decision Process

MSBE Mean-Squared Bellman Error

MAE Mean Absolute Error

MSSE Mean Steady-State Error

IRL Inverse Reinforcement Learning

Acknowledgements

The final outcome of this work required careful guidance from Paul-Arthur Dreyfus and Prof. Alireza Karimi, to whom I am sincerely grateful. Their assistance came in times troubled by the COVID-19 outbreak, during which they showed extreme flexibility and engagement. The consistent bi-weekly online meetings where a precious source of supervision and help.

This work drove me to explore fields that I was eager to discover, with great freedom. I am sure that thanks to this opportunity, I will continue on my journey to understand and apply artificial intelligence to challenging problems.

1 Introduction

Reinforcement Learning (RL) tackles the challenge of creating agents that can learn to perform a task in a dynamic environment, as opposed to following a pre-programmed behavior in static environments [1]. This field of Artificial Intelligence (AI) has grown in interest in recent years as it has been able to demonstrate promising and new approaches to solve difficult sequential decision making problems in fields reaching from natural and social sciences, to robotics and game control. Recent breakthroughs, that include mastering the game of Go [2] and outperforming the human Atari benchmark [3], have elevated the expectations in that field to new heights.

To explain the recent rise of RL one must express the importance of the role that Deep Learning (DL), and Deep Neural Networks (DNNs) in general, has played. The groundbreaking results mentioned leveraged the representation and abstraction power of DNNs in their novel architectures. This of course was enabled thanks to the rise of computing power available. The generality and flexibility that comes with end-to-end approaches enabled by Deep RL (DRL), namely the automatic feature engineering and distributed representational power, has also hampered the understanding of the models trained, which is essential to identify their strength and weaknesses in a design loop aimed at improving those systems.

The work presented in this report is a case study on State-Of-The-Art (SOTA) DRL algorithms applied to the position control of *Quanser*'s 3 DOF Gyroscope [4]. Gyroscopes can be used as attitude control devices in sea, air and space vehicles. Such technologies can also be leveraged as sensors to measure orientation and angular velocity. From a control standpoint, gyroscopes are interesting to study as they offer highly non-linear and coupled dynamics. Previous semester projects have tackled the challenge of controlling the gyroscope through Feedback Linearization (FL) and Multi-Input Multi-Output controller design (MIMO) [5] as well as using sliding mode control with model parameters identified with DNNs [6]. Similar research directions can be found in [7] and [8] where efforts are made to design adaptive controllers that are able to compensate for the modeling errors.

This system is thus chosen as the object of the case study in order to explore the DRL workflow for control, while maintaining a critical view on the strengths, weaknesses and interpretability of the agents that are trained. Since alleviating the issues posed by modeling errors in classical controllers is an important issue addressed by the works mentioned above, this case study also offers an interesting look at alternative ways to design controllers with an imperfect knowledge of the system at hand. The hierarchical objectives of this project are:

1. Train an agent that can achieve position tracking on the gyroscope with performance comparable to existing controllers.
2. Study the adaptability of the agent to the change in parameters and environmental dynamics.
3. Draw links and maintain a close parallel with classical control theory.
4. Interpret the agent's learned behavior.

To achieve these goals, a preliminary study on RL and on SOTA DRL algorithms in particular is made in Chapter 2 and in Chapter 3. The dynamics of the gyroscope are then derived and incorporated in a simulation of the environment compatible with standard RL frameworks in Chapter 4. A study on reward functions is done in Chapter 5 to engineer an appropriate reward function for the system. An agent is trained on a simplified model of the gyroscope through hyperparameter search in Chapter 6, followed by domain adaptation of that agent to make it robust to uncertainty in the model in Chapter 7. Interpretability of the agents is studied in Chapter 8. Finally, further research directions are outlined in Chapter 9.

Since this report covers both RL and control theory topics, a clarification on analogous terms needs to be done. An *agent* is a learning and control algorithm designed using RL, while a *controller* will be used to designate a system derived using standard control theory. An *action* taken according to a *policy* in RL is analogous to a *control signal* or to an *input* in standard control theory.

2 Reinforcement Learning Background

This chapter provides an overview of the essential concepts and notations used in the RL framework. The quantities describes here will be referred to throughout the whole length of this report. Most of the theory described here has been extracted from Open AI's *Spinning Up* framework documentation [9] and from R. Sutton and A. Barto's book [10]. The previous Master thesis on RL supervised by Paul-Arthur Dreyfus [11] has also served as a great summary.

2.1 Formalism

RL is an approach to tackle the sequential decision making problem in AI. The standard RL problem consists of an agent interacting with an environment E: it receives an observation o_t of it's true state s_t at time-step t, reacts by taking an action a_t, and receives a scalar signal r_t called a reward, as shown in Figure 1. The reward gives a feedback to the agent of the quality of the decision it made. Rewards can be extrinsic (generated by the environment) or intrinsic (an agent's motivation to discover for instance).

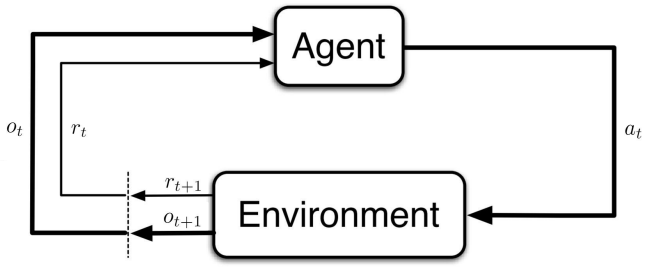

Figure 1: RL interaction schematic [10]

In the RL framework, the environment is modelled as a Markov Decision Process (MDP) $\mathcal{M}$ which consists of a tuple: $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}_{s'}^{sa}, \gamma, r \rangle$ where:

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $\mathcal{T}_{s'}^{sa}$ is the transition dynamics, if the environment is stochastic then $\mathcal{T}_{s'}^{sa} = P(s'|s, a)$.
- $\gamma \in [0, 1]$ is the discount factor, that scales the importance of future rewards and is necessary for convergence of the algorithms: with $\gamma = 0$, the agent considers only immediate rewards.
- r is the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

In a MDP, the transition dynamics satisfy the Markov property: $P(s_{t+1}|a_t, s_t, \ldots, a_0, s_0) = P(s_{t+1}|a_t, s_t)$. This property implies that the present state and corresponding action holds all the necessary information to predict the future state of the environment.

When the agent has access only to a partial observation o_t of the true state s_t of the environment, the MDP is known as being partially observable (POMDP). This is the case for the gyroscope and for most robotic applications, as all necessary information to predict accurately the future state of the system are not known, most notably the exact dynamical properties of the system at time-step t. However as an approximation, the standard MDP framework for RL is used. In the literature, this approximation is often implied and notations were it would be more appropriate to use o_t, such as when it is said that the action is conditioned by the observation of the environment, it is replaced directly by s_t even if the agent does not have direct access to the state but to the observation instead.

In the derivations of the RL background theory, we will use this notation and drop the use of o_t in favor of s_t, which will be more familiar for the reader. In the rest of the report however, we will use the more accurate notation o_t to define the observation that is returned by the Gym environment to the agent.

An agent's behavior is dictated by a policy $\pi(a|s)$ and defines a probability distribution over the action space $\mathcal{A}$ given the state s:

$$a_t \sim \pi(\cdot\,|s_t)$$

It can also be a deterministic rule in which case the notation can be substituted to avoid confusion:

$$a_t = \mu(s_t)$$

The multi-step return R_t is the discounted, accumulated reward overtime starting from time-step t onwards:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

The expected return taking into account the stochasticity in the environment along the trajectory τ that is taken (the sampling over the transition dynamics is implied and is not written), and the stochasticity in the policy, is given by:

$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R_0|\pi] = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t|\pi\right]$$

The goal of the RL problem is to find the policy π^* that maximizes the expected return $J(\pi)$ of the agent:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

2.2 Value Functions

Value functions are measures of how good a particular state, or state-action pair, is. They are at the core of RL techniques as optimization algorithms will make use of them in order to orient the agent towards decisions that will lead it to states or state-action pairs with high value.

The two main value functions are the state value function V_π that measures the value of a particular state, and the action value function (Q-value function or Q-function) Q_π that measures the value of a particular state-action pair. From now on, the dependency on the policy is implied using the superscript $^\pi$:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R_t|s = s_t]$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R_t|s = s_t, a = a_t]$$

These functions reflect the expected return the agent will get by starting at state s (or starting at state s and taking action a), and following the policy π onwards.

In particular, the optimal value functions can be decomposed under the Bellman equation where the right hand side is the famous Bellman backup operator:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim P}}[r(s,a) + \gamma V^\pi(s')]$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[r(s,a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}[Q^\pi(s',a')]\right]$$

In the above formulas, the expectancy is taken by sampling over: $s' \sim P(\,\cdot\,|s,a)$, $a \sim \pi(\,\cdot\,|s)$ and $a' \sim \pi(\,\cdot\,|s')$. The Bellman equation thus relates the values of consecutive states linked by an action taken under the policy and transition dynamics.

If the agent behaves according to the optimal policy π^* of the problem, the value functions yield the optimal value functions:

$$V^*(s) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R_t|s = s_t]$$

$$Q^*(s,a) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R_t|s = s_t, a = a_t]$$

In particular, the optimal value functions can be decomposed under the Bellman equation:

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P}[r(s,a) + \gamma V^*(s')]$$

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

In the above formulas, the expectancy is taken by sampling over: $s' \sim P(\,\cdot\,|s,a)$.

Variants of these formulas are used throughout the RL literature and are at the core of the algorithms, as the value functions can be used to obtain a policy. In fact, if the optimal value functions are found, the optimal policy can be extracted by acting greedily on them,

like the max operator above shows. If the transition dynamics are known, the optimal policy derives from acting greedily with respect to the optimal state value function. The optimal policy can also be found by acting greedily with respect to the Q-function, without the need of knowing the transition dynamics:

$$a^*(s) = \arg\max_a Q^*(s, a)$$

This last point is why many algorithms favor the Q-function. The state value can still be used as a baseline in the case of unknown dynamics for policy gradient algorithms, which will be described in Chapter 2.7.

2.3 Model-Based and Model-Free RL

If the full MDP is known, dynamic programming can be used in order to iterate over the Bellman equation in algorithms called value iteration and policy iteration. Both algorithms are proved to converge to the optimal value and policy [10]. Those approaches are called *planning*.

If the MDP is not known, it can be discovered in methods known as *model-based*. On the other hand, *model-free* approaches assume no knowledge of the model and only learn from experience, they are thus less sample-efficient as they do not construct a model of the environment (at least not explicitly). In this project, the goal is to learn a policy that can assume as little knowledge as possible from the dynamics of the gyroscope in order to study if the issues posed by the modeling errors in traditional control approaches can be alleviated. Thus, the project will make use of model-free algorithms, and model-based ones will not be described in more details.

In model-free problems, the transition dynamics are unknown and thus planning is not possible, and the Bellman equations cannot be used as such. The two main approaches to this problem are Monte Carlo methods (MC) and Temporal Difference learning (TD), which iteratively build estimates of the value functions by interacting with the environment.

2.4 Exploration and Exploitation

Before jumping into the details of model-free approaches, a point has to be made on the exploration-exploitation trade-off. In fact, model-free algorithms build empirical estimates of the value functions by exploring the environment and collecting reward information when encountering states and when performing actions. A way to picture this would be to build lookup tables for states (when estimating the state value function) or for state-action pairs (when estimating the Q-function) and to iteratively update the estimates of the value functions when encountering corresponding states or state-action pairs in the lookup tables.

The agent will start off with a random policy and explore the environment. In the case of dynamic programming (when the full MDP is known), the actions are taken greedily with respect to the current estimates of the state value function or the Q-function, and following the optimal Bellman equation seen above. It is proved that when the MDP is known, this approach is guaranteed to converge. In model-free however, the estimates are empirical and thus suffer from local optimas. The actions taken in the environment thus cannot

be greedy with respect to the current lookup table, but ϵ-greedy over the Q-values in order to allow exploration. Other strategies can be implemented, such as a softmax over Q-values.

Thus, the classic problem of exploitation and exploration arises, and tuning must be done in order to prevent the agent from falling into a local optima (exploration), while making sure that it converges (exploitation).

2.5 Monte Carlo and TD learning

Monte Carlo methods do not assume any knowledge of transition dynamics, and construct estimates of the value functions by relying on empirical means across many trials. By the law of large numbers the value functions will converge to the true value functions. This method is highly empirical and can only be applied to episodic tasks (where the infinite sum is reduced to a finite one).

TD learning, on the other hand, uses the Bellman equation in order to refine the estimate of the value functions across the interactions with the environment, by bootstrapping. Bootstrapping means estimating value functions at a state or state-action pair based on the subsequent states encountered. The update rule for TD learning on the state value function using the Bellman equation is:

$$V(s) \leftarrow V(s) + \alpha \left[r(s,a) + \gamma V(s') - V(s) \right]$$

To estimate the Q-function, there are two categories of update rules:

- SARSA:
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r(s,a) + \gamma Q(s',a') - Q(s,a) \right]$$
- Q-learning:
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

The difference in those two algorithms is the way the action a' from the state-action pair (s', a') that is used to bootstrap $Q(s,a)$ is taken. In SARSA algorithms, a' is taken according to the current policy π, whereas in Q-learning, a' is taken by acting ϵ-greedily with respect to the Q-values. The key difference is that SARSA is an *on-policy* update rule whereas Q-learning is *off-policy*, meaning that $Q(s,a)$ is bootstrapped to a state-action pair where the action a' was not derived using the current policy. The importance of such a distinction will become important in Chapter 3.1.

2.6 Function Approximations

In Chapter 2.4, the naive approach of the lookup table was mentioned in order to store the current estimates of the value functions. This approach can work for very simple problems with a discrete state and action space, but the approach has to be generalized to large or continuous state and action spaces. In order to do so, function approximators that generalize from examples, and which are parametrized by $\boldsymbol{\phi}$, can be used. Many function approximators can be used, such as radial basis function networks, or DNNs tuned by backpropagation in the case of DRL. These regressors can then be tuned in a supervised learning approach by

gradient descent on a loss function. When estimating value functions, the loss would then be the TD learning update rules seen above. For the Q-value, the loss for a set $\mathcal{D}$ of collected transitions (s, a, r, s'), also called the Mean-Squared Bellman Error (MSBE), is:

$$L(\phi, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim\mathcal{D}} \left[\frac{1}{2} \left[r + \gamma \max_{a'} Q_\phi(s', a') - Q_\phi(s, a) \right] \right]$$

Gradient descent on this loss yields:

$$\phi \leftarrow \phi - \alpha \nabla_\phi L(\phi, \mathcal{D})$$

The MSBE characterizes the accuracy of the current state of the Q-value network, by evaluating the match to the optimal Bellman equation. Performing gradient descent directly on the loss above would be unstable. The part $y(r, s') = r + \gamma \max_{a'} Q_\phi(s', a')$ is called the target, and arises from the temporal difference backup. The problem is that, during learning, the target for $Q(s, a)$ also depends on the current parametrization of the Q-value regressor. Performing gradient descent as such would lead to divergence. It is thus common practice to perform strategies that stabilize the learning by making a copy of the regressor, which would then be called the target regressor, and updating it at a lower pace to prevent those instabilities.

2.7 Q-learning and Policy Optimization

In Chapter 2.2, for the model-free case, the policy was said to be derived from the Q-values by taking:

$$a^*(s) = \arg\max_a Q^*(s, a)$$

These types of algorithms form the Q-learning family. Algorithms like DQN build Q-value regressors using DNNs, that map a continuous (and/or very large and/or discrete) state space to the Q-values of a discrete action space. The output of the DNN is thus multidimensional and contains the Q-values of all the action space. The policy is then derived from such a network by acting greedily using the max operator. For continuous action spaces, DQN could be used by discretizing the action space.

In practice however, DQN becomes intractable if the discretized action space is very large. Policy optimization methods offer thus an alternative by directly learning a policy π. To do so, the policy is parametrized by a set of parameters θ using a function approximator π_θ. The parameters are then tuned by maximizing the expected return objective function:

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} [R_0|\pi_\theta] = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi_\theta \right]$$

Gradient ascent on this objective function yields:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

For a stochastic policy $\pi(a|s)$, the gradient can be shown by the stochastic policy gradient theorem [12] to be equal to:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{a'\sim\pi} [\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)]$$

For a deterministic policy $\mu(s)$, the gradient can be shown by the deterministic policy gradient theorem [13] to be equal to:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right]$$

2.8 Actor-Critic Architecture

The policy gradient theorems can be used in the tuning process of the regressor used, such as backpropagation for DNNs, to perform gradient ascent in order to maximize the objective function. Policy optimization allows to directly tune the policy, instead of deriving it by acting greedily on the Q-values. However, we see that in both policy gradients, the Q-function is needed. In simpler algorithms, like *Reinforce*, the returns R_t can be used as an unbiased sample of the Q-values. It can be shown that subtracting a baseline to the returns, such as a running average, can reduce the variance and thus stabilize the learning, without changing the gradient in expectation. This approach is however still noisy. An alternative would be to estimate the Q-value using a regressor and a loss as well. Such a regressor is called a *critic*. Policy optimization methods using a critic are called *actor-critic* methods. Other value functions can be used, such as the advantage function: $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$.

3 Algorithms

Of the various types of model-free RL architectures presented in Chapter 2, namely Q-learning, policy optimization and actor-critic, many different algorithms exist. In this project, we focus on three SOTA actor-critic algorithms leveraging DNNs as function approximators: the *Deep Deterministic Policy Gradient* (DDPG) [14], the *Twin Delayed DDPG* (TD3) [15] and the *Soft Actor Critic* (SAC) [16] algorithms. In this case study, we decide to focus on deterministic algorithms (DDPG and TD3), yet we also take a look at SAC.

3.1 DDPG

DDPG is an actor-critic algorithm using DNNs as function approximations and thus learns: a policy network paramterized by θ for the actor and a Q-value network parametrized by ϕ for the critic. It concurrently learns a deterministic policy $\mu_\theta(s)$ and $Q_\phi(s,a)$. As the agent interacts with the environment, it collects a set $\mathcal{D}$ of transitions (s,a,r,s').

On the policy optimization side of DDPG, the goal is to perform backpropagation on the actor network using the deterministic policy gradient. The expectation in the deterministic policy gradient can be approximated by a sample mean on a minibatch B of transitions sampled from $\mathcal{D}$. The deterministic policy gradient then performs a stochastic gradient ascent on the objective function using the following estimate of the gradient:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{|B|} \sum_{s \in B} \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}$$

On the Q-learning side of DDPG, the goal is to perform backpropagation on the MSBE loss, characterizing the accuracy of the current state of the Q-function regressor by bootstrapping. On the minibatch B, this gives:

$$\nabla_\phi L(\phi, B) = \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_\phi(s,a) - y(r,s'))^2$$

As discussed in Chapter 2.6, computing the targets $y(r,s')$ with the parametrization $Q_\phi(s,a)$ of the Q-function would be unstable. Thus a separate critic network is used, the target critic network, and parametrized separately by ϕ_{targ}. Furthermore, in order to act greedily on the target critic network, the following approximation is made:

$$\max_{a'} Q_{\phi_{\text{targ}}}(s',a') \approx Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

This approximation, which is valid in the limit case where the derived policy is the optimal policy, works if a target policy network $\mu_{\theta_{\text{targ}}}(s')$ is used. In order to let the two target networks evolve at a lower pace compared to the other networks, polyak averaging is used:

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

In the update rules above, ρ is the polyak interpolation factor. The set $\mathcal{D}$ from which we sample the minibatch B for the stochastic gradient descent is called the replay buffer.

It stores all the transition (s, a, r, s') encountered when interacting with the environment. Since all transitions are stored (even those acquired using an old state of the actor network), DDPG is an off-policy algorithm. The reason that it is possible to do so, is that the optimal Bellman equation used in the MSBE should be satisfied for all state transitions. Since DDPG uses a deterministic policy, exploration is not built-in explicitly. Thus, noise is added artificially during training: $a = \text{clip}(\mu_\theta(s), +\epsilon, a_{Low}, a_{High})$ where $\epsilon \sim \mathcal{N}$ (Gaussian noise has been found to work well). The variance of the Gaussian can be decayed throughout the training to help convergence and to get higher quality samples, but this is not the case in the Spinning Up library. The hyperparameters are:

- Epochs: the number of epochs to train the agent.
- Episode length: the length of a simulation episode, which is fixed during training in this work. Longer episodes will make training slower, but will represent more accurately the infinite-horizon returns.
- Steps per epoch: the number of interactions between the environment and the agent per epoch, equal to the product of an episode length and the number of episodes per epoch.
- γ: the discount factor, in the range $[0, 1]$. A discount factor close to 1 will give more importance to future rewards.
- π learning rate: the learning rate of the actor network. With a small learning rate, the DNN will converge slower and might get stuck in local optimas. With a large learning rate, the DNN may be more robust to local optimas but may not converge.
- Q learning rate: the learning rate of the critic network. With a small learning rate, the DNN will converge slower and might get stuck in local optimas. With a large learning rate, the DNN may be more robust to local optimas but may not converge.
- ρ: the polyak interpolation factor, in the range $[0, 1]$. Controls the speed of the update of the target networks.
- Batch size: the minibatch size for the stochastic gradient descent. Large batch sizes may be more stable, but harder to compute.
- Start steps: number of initial steps performed with a random policy, before switching to the real policy. Trick used to help exploration at the beginning.
- Hidden sizes: The number and size of the hidden layers of the DNN. Deep networks may be able to capture more complex features but will be harder to tune.
- Actor noise: the standard deviation of the Gaussian noise added to the actor during training. Noisier policies will explore more, but at the cost of possibly not converging.
- Replay size: size of the replay buffer.

3.2 TD3

TD3 has been developed in order to alleviate some of the issues of DDPG. Most notably, the policy has been found to exploit over-estimations of the Q-value regressor, leading to a failure. To solve this, TD3 adds three features to DDPG:

1. Clipped double-Q learning: two different DNNs are used, to learn concurrently two different Q-value function approximation $Q_{\phi,1}(s,a)$ and $Q_{\phi,2}(s,a)$, each having their own separate target critic network $Q_{\phi_{\text{targ},1}}(s', \mu_{\theta_{\text{targ}}}(s'))$ and $Q_{\phi_{\text{targ},2}}(s', \mu_{\theta_{\text{targ}}}(s'))$. During their training, a single target $y(r,s')$ is used, which is computed using the minimum value computed by the two target critic networks. This way, punctual over-estimations cannot be exploited as the minimum value of the two will be used.

2. Delayed policy updates: the policy and target policy networks are updated at a smaller frequency than the Q-value regressors, to cutoff the use of Q-value over-estimations.

3. Target policy smoothing: clipped noise is added to the output of the target policy $\mu_{\theta_{\text{targ}}}(s')$ in order to smooth out abrupt changes in policy. The policy tends to exploit over-estimations in the Q-function, and can thus result in abrupt changes in policy at some actions. By adding noise on the target action, sharp peaks will tend to be avoided by picking actions in the neighborhood.

The added hyperparameters are:

- Policy delay: the number of critic update between each policy update.
- Target noise: the standard deviation of the noise added to the target policy to smooth out sharp policy changes.
- Noise clip: limit bounds for the target policy smoothening noise.

Thus, TD3 offers a promising alternative as it is less optimistic with regards to the Q-values and more prudent, without diverging significantly from DDPG in terms of structure and added hyperparameters.

3.3 SAC

Compared to TD3, SAC diverges more from DDPG, and so it was not studied in as much details. Still, it was experimented with in order to showcase some of it's strength and attributes. SAC learns a stochastic policy, thus explicitly incorporating exploration as opposed to DDPG and TD3. Exploration is controlled in SAC through *entropy regularization*, which adds an entropy term to the returns, changing the objective function of the policy gradient to feature this trade-off between maximizing the returns and the entropy which reflects the randomness of the policy. Exploration is thus embedded in the objective function, which can help the agent avoid local optimas. The hyperparameters are not described here.

3.4 Implementation

To train an agent on the gyroscope Gym environment using these algorithms, many different Python frameworks are available, namely Open AI's *Spinning Up*, Berkeley's *RLlib* and *Stable Baselines*. Spinning Up was chosen early on for it's ease of use, thorough documentation and convenient plotting functions. However it was found that the simplicity of the framework, while convenient in order to get started fast, was ultimately detrimental. Most notably, Spinning Up's implementations of DDPG, TD3 and SAC do not support parallelization yet. Also, scheduled noise and learning rates cannot be implemented. For these reasons, tackling RLlib's steeper learning curve from the start would be rewarding for the reader instead of going forward with Spinning Up.

4 Environment

In order to train a RL agent to control the gyroscope, a naive approach would be to train it on the actual setup, to obtain true physical samples. However such an approach would be unpractical as training would be unrealistically long, and pose security issues since the agent will explore in the early stages of training, resulting in dangerous behaviors. Therefore, as in most RL works, a simulation of the physical system is leveraged in order to accelerate training. This raises the question of the aptitude of the policy learned in simulation to transfer to the physical system. Performing an extensive analysis on the accurate modeling of the gyroscope to build a high-fidelity simulator would alleviate such an issue. However, whether the policy will be able to perform well on the physical system is still not guaranteed. In addition, such an approach would question the relevance of a RL approach to the control problem, as an accurate model of the system could directly be used with classical control methods that have already proven to be effective. Thus, the model used in the simulation will be an approximate representation of the dynamics of the system, and further work will be done in Chapter 7 to tackle the problem of the simulation to reality transfer.

4.1 Modeling

The environment that the RL agent will interact in will therefore be a numerical simulation of the Quanser 3 DOF gyroscope. This 3 DOF system has two gimbals, an outer red one and an inner blue one, described by the angular positions θ and ϕ, as well as a rotating gold disk of angular position $\dot{\psi}$. In order to have reasonable training times and to avoid making the simulation a speed bottleneck, the disk angular velocity $\dot{\psi}$ is assumed to be constant and will be set to $\dot{\psi} = \omega$. This is a reasonable approximation since the disk angular velocity is usually controlled separately using standard PID controllers.

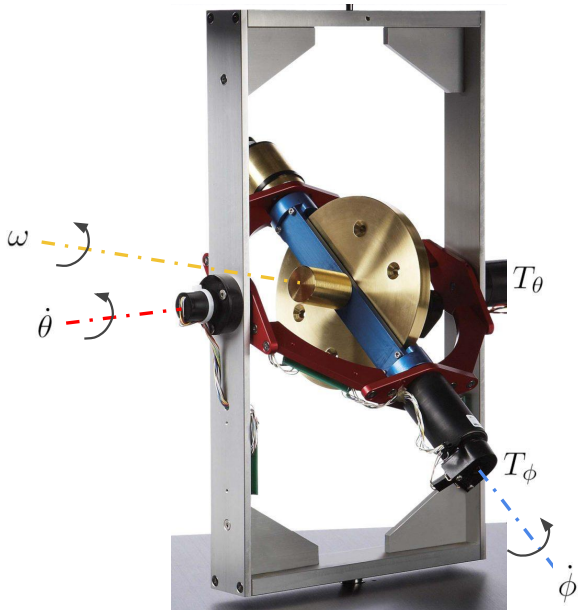

Figure 2: Quanser's 3 DOF Gyroscope

The equations of motion of the gyroscope can be derived by using Lagrangian mechanics. The 2 generalized coordinates of the gyroscope are $q_i = \{\theta, \phi\}$ and thus 2 dynamical equations will be derived. Throughout the derivations, the subscripts $_r$ and $_b$ will be used to characterize a physical quantity of the red or the blue gimbal respectively. The subscripts $_\theta$ and $_\phi$ will be preferred when expressing the control torques T and input voltages u of the gimbals. The Euler-Lagrange equations give:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \frac{\partial \mathcal{L}}{\partial q_i} = Q_i$$

The rotational energy is the only component of the Lagrangian:

$$\mathcal{L} = \frac{1}{2}(\boldsymbol{\omega}_r^T \mathbf{J}_r \boldsymbol{\omega}_r + \boldsymbol{\omega}_b^T \mathbf{J}_b \boldsymbol{\omega}_b + \boldsymbol{\omega}_d^T \mathbf{J}_d \boldsymbol{\omega}_d)$$

Where:

- $\mathbf{J}_i$: the inertia tensor of the i^{th} body expressed in its inertial frame
- $\boldsymbol{\omega}_i$: the rotational vector of the i^{th} body expressed in its inertial frame

In particular, the inertia tensors are:

$$\mathbf{J}_r = \begin{bmatrix} J_{rx_1} & 0 & 0 \\ 0 & J_{rx_2} & 0 \\ 0 & 0 & J_{rx_3} \end{bmatrix} \quad \mathbf{J}_b = \begin{bmatrix} J_{bx_1} & 0 & 0 \\ 0 & J_{bx_2} & 0 \\ 0 & 0 & J_{bx_3} \end{bmatrix} \quad \mathbf{J}_r = \begin{bmatrix} J_{dx_1} & 0 & 0 \\ 0 & J_{dx_2} & 0 \\ 0 & 0 & J_{dx_3} \end{bmatrix}$$

For the rotational vectors, we follow the geometrical derivations found in [5], and obtain:

$$\boldsymbol{\omega}_r = \begin{bmatrix} \dot{\theta} \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{\omega}_b = \begin{bmatrix} \cos(\phi)\dot{\theta} \\ \dot{\phi} \\ \sin(\phi)\dot{\theta} \end{bmatrix} \quad \boldsymbol{\omega}_r = \begin{bmatrix} \cos(\psi)\sin(\phi)\dot{\theta} + \sin(\omega t)\dot{\phi} \\ -\cos(\phi)\sin(\psi)\dot{\theta} + \cos(\omega t)\dot{\phi} \\ \omega + \sin(\phi)\dot{\theta} \end{bmatrix}$$

The Lagrangian is then equal to:

$$\mathcal{L}(\dot{\theta}, \phi, \dot{\phi}) = \frac{1}{4}\Big\{2(J_{bx_1} + J_{dx_1})\dot{\phi}^2 + 2J_{dx_3}\omega^2 + 4J_{dx_3}\sin(\phi)\dot{\theta}\omega$$
$$+(J_{bx_1} + J_{bx_3} + J_{dx_1} + J_{dx_3} + 2J_{rx_1} + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\cos(2\phi))\dot{\theta}^2\Big\}$$

For the friction-less gyroscope and by introducing the generalized forces corresponding to the torques T_θ and T_ϕ applied to the gimbals, the equations of motion are then derived from the following system:

$$\begin{cases} \dfrac{d}{dt}\dfrac{\partial \mathcal{L}}{\partial \dot{\theta}} - \dfrac{\partial \mathcal{L}}{\partial \theta} = T_\theta \\ \dfrac{d}{dt}\dfrac{\partial \mathcal{L}}{\partial \dot{\phi}} - \dfrac{\partial \mathcal{L}}{\partial \phi} = T_\phi \end{cases}$$

Using *Mathematica*, the equations lead to 2 non-linear coupled differential equations:

$$\frac{1}{2}(J_{bx_1} + J_{bx_3} + J_{dx_1} + J_{dx_3} + 2J_{rx_1} + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\cos(2\phi))\ddot{\theta}$$
$$+J_{dx_3}\omega(\phi)\dot{\phi} - (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sin(2\phi)\dot{\phi}\dot{\theta} = T_\theta$$

$$(J_{bx_2} + J_{dx_1})\ddot{\phi} + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\cos(\phi)\sin(\phi)\dot{\theta}^2 - J_{dx_3}\omega\cos(\phi)\dot{\theta} = T_\phi$$

Before solving the equations for $\ddot{\theta}$ and $\ddot{\phi}$, the following change of variable will be made, which is more familiar in a state-space representation context:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \\ \phi \\ \dot{\phi} \end{bmatrix}$$

where $\boldsymbol{x}$ is the standard state vector of the state-space representation, not to be confused with the observation o_t and the state s_t of RL. Finally, solving for $\dot{x}_1, \dot{x}_2, \dot{x}_3$ and $\dot{x}_4$, the state-space system of equations is:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \dfrac{T_\theta + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sin(2x_3)x_2x_4 - J_{dx_3}\cos(x_3)x_4\omega}{J_{bx_1} + J_{dx_1} + J_{rx_1} + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sin^2(x_3)} \\ \dot{x}_3 = x_4 \\ \dot{x}_4 = \dfrac{T_\phi - (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\cos(x_3)\sin(x_3)x_2^2 + J_{dx_3}\cos(x_3)x_2\omega}{J_{bx_2} + J_{dx_1}} \end{cases}$$

For the gyroscope with friction, the generalized forces in the Euler-Lagrange system of equations are changed to incorporate static and dynamic friction:

$$\begin{cases} \dfrac{d}{dt}\dfrac{\partial \mathcal{L}}{\partial \dot{\theta}} - \dfrac{\partial \mathcal{L}}{\partial \theta} = T_\theta - f_{vr}\dot{\theta} - f_{cr}\,\mathrm{sign}(\dot{\theta}) \\ \dfrac{d}{dt}\dfrac{\partial \mathcal{L}}{\partial \dot{\phi}} - \dfrac{\partial \mathcal{L}}{\partial \phi} = T_\phi - f_{vb}\dot{\phi} - f_{cb}\,\mathrm{sign}(\dot{\phi}) \end{cases}$$

Following the same procedure as for the friction-less case, the state-space system of equations for the gyroscope with friction is:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \dfrac{T_\theta - f_{vr}\dot{\theta} - f_{cr}\,\mathrm{sign}(\dot{\theta}) + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sin(2x_3)x_2x_4 - J_{dx_3}\cos(x_3)x_4\omega}{J_{bx_1} + J_{dx_1} + J_{rx_1} + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sin^2(x_3)} \\ \dot{x}_3 = x_4 \\ \dot{x}_4 = \dfrac{T_\phi - f_{vb}\dot{\phi} - f_{cb}\,\mathrm{sign}(\dot{\phi}) - (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\cos(x_3)\sin(x_3)x_2^2}{J_{bx_2} + J_{dx_1}} \\ \qquad\qquad + \dfrac{J_{dx_3}\cos(x_3)x_2\omega}{J_{bx_2} + J_{dx_1}} \end{cases}$$

The control inputs to the gyroscope are the input voltages to the DC motors generating the torques T_θ and T_ϕ. We thus need to express the torques T_θ and T_ϕ as a function of the input voltages u_θ and u_ϕ to the DC motors controlling the two gimbals. The dynamics of the motors are faster than the mechanical dynamics of the gyroscope. Therefore, the circuit model of the DC motors can be simplified to a simple DC motor steady-state gain K_T. The torques can thus be expressed as the product of the current gain K_{Amp}, the DC motor

steady-state gain K_T, the motor efficiency η, the gearbox ratio n_i and the input voltage u_i:

$$\begin{cases} T_\theta = (K_{Amp} K_T \eta n_r) u_\theta \\ T_\phi = (K_{Amp} K_T \eta n_b) u_\phi \end{cases}$$

The numerical values used throughout the project for the motor and gyroscope constants are resumed in Table 1 and Table 2 respectively (see [4] for datasheets).

Table 1: Motor constants

Quantity	Value	Symbol	Dimension
Current Gain	0.5	K_{Amp}	$\mathrm{A\,V^{-1}}$
Motor Gain	0.0704	K_T	$\mathrm{N\,m\,A^{-1}}$
Motor Efficiency	0.86	η	1
Red Gearbox Ratio	1.5	n_r	1
Blue Gearbox Ratio	1	n_b	1

Table 2: Gyroscope constants

Quantity	Value	Symbol	Dimension
Red Inertia $\boldsymbol{e}_1$	0.0179	J_{rx_1}	$\mathrm{kg\,m^2}$
Blue Inertia $\boldsymbol{e}_1$	0.0019	J_{bx_1}	$\mathrm{kg\,m^2}$
Blue Inertia $\boldsymbol{e}_2$	0.0008	J_{bx_2}	$\mathrm{kg\,m^2}$
Blue Inertia $\boldsymbol{e}_3$	0.0012	J_{bx_3}	$\mathrm{kg\,m^2}$
Disk Inertia $\boldsymbol{e}_1$	0.0028	J_{dx_1}	$\mathrm{kg\,m^2}$
Disk Inertia $\boldsymbol{e}_2$	0.0056	J_{dx_2}	$\mathrm{kg\,m^2}$
Disk Inertia $\boldsymbol{e}_3$	0.0056	J_{dx_3}	$\mathrm{kg\,m^2}$

4.2 Feedback Linearization

In order to compare the agent that we will derive throughout this project to a baseline, the controller derived in [5] through FL will be implemented. The purpose of implementing such a controller is also to evaluate the accuracy of the simulation created in this project, as it will be shown in the Chapter 4.3.

FL is a technique in which a feedback law and a transformation are computed in order to convert a non-linear system into a linear and controllable one. The details can be found

in [5], and yield the following control law:

$$
\begin{aligned}
u_\theta =& \frac{1}{2}\Big\{(J_{bx_1} + J_{bx_3} + J_{dx_1} + J_{dx_3} + 2J_{rx_1} + J_{bx_1}\cos(2x_3) - J_{bx_3}\cos(2x_3) + J_{dx_1}\cos(2x_3) \\
& - J_{dx_3}\cos(2x_3))\sigma_\theta 2x_{1,ref} + 2\omega J_{dx_3}\sigma_\omega \sin(x_3) - \sigma_\theta 2(J_{bx_1} + J_{bx_3} + J_{dx_1} + J_{dx_3} + 2J_{rx_1} \\
& + (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\cos(2x_3))x_1 + 2x_2(f_{vr} - J_{bx_1}\sigma_\theta - J_{bx_3}\sigma_\theta - J_{dx_1}\sigma_\theta \\
& - J_{dx_3}\sigma_\theta - 2J_{rx_1}\sigma_\theta - (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sigma_\theta\cos(2x_3) \\
& - (J_{bx_1} - J_{bx_3} + J_{dx_1} - J_{dx_3})\sin(2x_3)x_4) - 2J_{dx_3}\sigma_\omega\sin(x_3)\omega + 2J_{dx_3}\cos(x_3)x_4\omega\Big\} \\
u_\phi =& -(J_{bx_2} + J_{dx_1})(-\sigma_\phi(x_{3,ref}\sigma_\phi - \sigma_\phi x_3 - 2x_4) \\
& + \frac{(-J_{bx_1} + J_{bx_3} - J_{dx_1} + J_{dx_3})\cos(x_3)\sin(x_3)x_2x_2 - f_{vb}x_4 + J_{dx_3}\cos(x_3)x_2\omega}{J_{bx_2} + J_{dx_1}}
\end{aligned}
\tag{1}
$$

Here: $-\sigma_\theta, -\sigma_\phi$ and $-\sigma_\omega$ are the poles chosen by the user.

4.3 Simulation

Having derived the simplified equations of motion of the gyroscope as well as the FL control law, the mechanical system can then be simulated in Python using the `integrate.solve_ivp` command from the *SciPy* library, which solves an initial value problem for a system of ODEs. The solver is chosen to be the explicit Runge-Kutta method of order 5 in this project.

Due to the COVID-19 outbreak, a proper simulation verification process to compare this method with actual experimental data from the gyroscope was not possible. Instead, in addition to the control of the general coherence of the simulation results, a simple verification was made by reproducing one experiment conducted in [5], where Y. Agram has done a proper simulation verification process during his project. Roughly, if the results of the experiment on the two different simulations are coherent, the simulation created here should carry the same attributes as in [5], and thus reflect the same similarities and differences with the experimental data. Since the simulation in this project is a simplification of the 3 DOF gyroscope to only 2 DOF, the results should also reveal the accuracy of such an assumption.

The same step tracking experiment as in [5] is thus conducted, for $\theta_{ref} = 45°$, $\phi_{ref} = -60°$ and $[\theta_0, \phi_0] = [0°, 0°]$ and at a disk angular velocity of $\omega = 300$ RPM. The same friction constants as in [5] are used. They are shown in Table 3.

Table 3: Friction constants

f_{cr}	f_{vr}	f_{cb}	f_{vb}
0	0.002679	0	0.005308

The plots for the angular positions and velocities in Figure 3 and 4 show very comparable results with the Figure 33 of [5], and thus conclude the simulation verification. For completeness, data of the gyroscope's angular position and velocities as well as control inputs should be acquired on the physical system for a variety of regimes and compared with the simulation that is fed with the same control inputs.

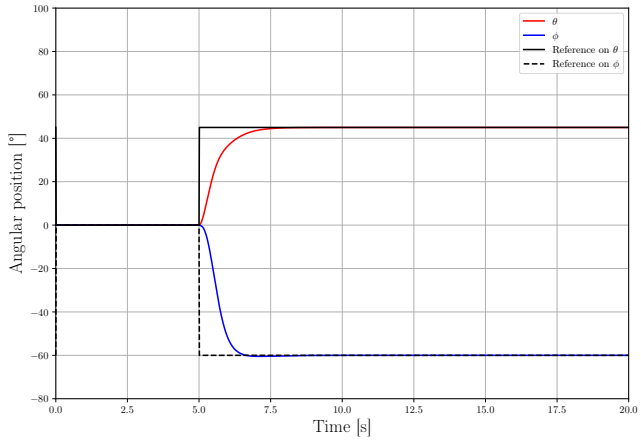

Figure 3: Step tracking angular positions on simulated gyroscope for $\theta_{ref} = 45°$ and $\phi_{ref} = -60°$ and $[\theta_0, \phi_0] = [0°, 0°]$ at 300 RPM

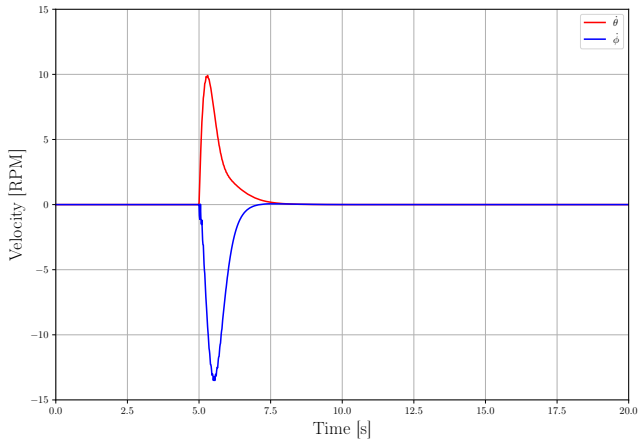

Figure 4: Step tracking velocities on simulated gyroscope for $\theta_{ref} = 45°$ and $\phi_{ref} = -60°$ and $[\theta_0, \phi_0] = [0°, 0°]$ at 300 RPM

4.4 Gym

Following previous projects, it was chosen early on to follow Open AI's *Gym* reinforcement learning framework for the gyroscope environment, which allows easy integration with many implementations of RL algorithms. Before doing so, the simulation must be adapted to the RL framework. The essential variables of the MDP are:

- The **state** $s_t = (\theta, \dot{\theta}, \phi, \dot{\phi}, \theta^{ref}, \phi^{ref}, \omega)$, a 7D continuous space.
- The **observation** $o_t = (\cos\theta, \sin\theta, \dot{\theta}, \cos\phi, \sin\phi, \dot{\phi}, \theta^{ref}, \phi^{ref}, \omega)$, a 9D continuous space normalized to $[-1, 1]$.
- The **action** $a_t = (u_\theta, u_\phi)$, a 2D continuous space normalized to $[-1, 1]$.
- The **reward** $r_{t+1} = \rho(s_{t+1}, a_t)$, where ρ is a function defined at training time.

The observation o_t, the action a_t and the reward r_{t+1} are fed to the DNN regressor during training. Empirical evidence shows that DNNs learn better when the data is normalized before using it as training samples. The observation o_t and the action a_t are thus normalized beforehand to boost the learning process. Special attention will be given to the scaling of the reward reward r_{t+1} in Chapter 5.2.

A Gym environment is a Python class with a set of methods that will be called by the RL algorithms during training to perform interactions. The essential methods for a Gym class to work are:

- `__init__`: to initialize the simulation parameters (inertias, motor constants, friction coefficients) as well as to define the observation space and the action space.
- `reset`: to reset the gyroscope to a random initial state, and returns an observation of that state.
- `step`: to perform a step in the simulation using the action taken by the agent. Runs the simulation for T_s and returns an observation of the new state of the system, the reward obtained and some optional information. T_s is the controller time-step, in this project we chose $T_s = 0.05s$ to obtain reasonable training times. During training, a complete simulation episode consists of 110 steps of length $T_s = 0.05s$.
- `render`: to plot or render the environment. Not used in this project.
- `close`: to close the existing environment.

Apart from these essential Gym methods, other methods were added to the environments created in this project, the most important of which are:

- `args_init`: to initialize the reward function chosen by the user at training time, as well as any other argument that cannot be passed through the standard `__init__` method.
- `_get_obs`: to transform the state of the environment into an observation vector.
- `reset2state`: to reset the environment to a desired initial position.
- `init_param`: to initialize the friction and noisy inertia parameters.

- `randomize_param`: to randomize the friction and noisy inertia parameters.

Apart from these methods, accessing methods as well as non-member methods were implemented to implement various tasks such as the reward functions to be called by `step`, the state-space model and normalizing functions.

4.5 Discontinuity

In a first iteration of this work, the observation returned by the Gym environment was defined to be the same as the state stored in the environment class:

$$o_t = s_t = (\theta, \dot{\theta}, \phi, \dot{\phi}, \theta^{ref}, \phi^{ref}, \omega)$$

This definition of the observation was chosen as it is often used in RL control problems involving angular positions [17]. However it was found that this definition is not prudent. To show why, we introduce a step tracking experiment for $\theta_{ref} = \pm 180°$ and $[\theta_0, \phi_0] = [0°, 0°]$ at a disk angular velocity of 200 RPM shown in Figure 5. This experiment was done using a DDPG agent trained on the environment with this wrong choice of the returned observation o_t.

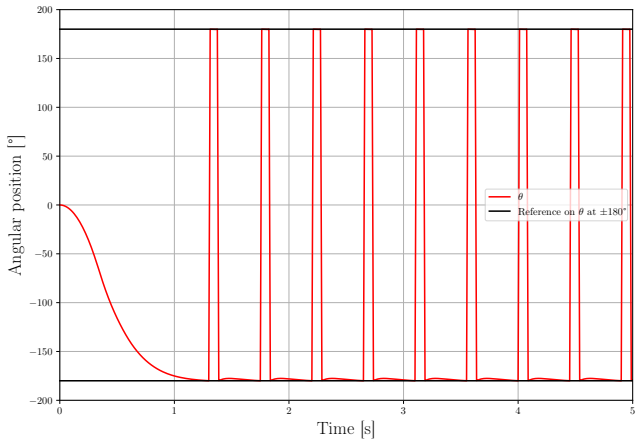

Figure 5: Step tracking on discontinuous environment for $\theta_{ref} = \pm 180°$ and $[\theta_0, \phi_0] = [0°, 0°]$ at 200 RPM

The resulting oscillating behavior is due to the angular positions being normalized to the interval $[-180°, 180°]$ (then normalized further to $[-1, 1]$ right before feeding it to the DNN to help the learning process as explained in Chapter 4.4) and to the definition of o_t. The normalization is necessary in order to keep the range of the observation space finite in the Gym environment. This however causes a discontinuity at the $-180°/180°$ boundary, which should actually be circular. Since the angular positions are used as such in the definition of the observation o_t, the DNN regressor learns function approximations with this discontinuity. The $-180°$ and $180°$ angular positions are thus "far" in terms of the DNN regressor

domain of definition. Throughout the learning process, the regression in the $-180°$ neighborhood will be more influenced by the $[-180°, 0°]$ range of the domain, while the $180°$ neighborhood will be more influenced by the $[0°, 180°]$ range of the domain. Since the behavior of the system will be similar in these two neighborhoods, the learned regressors will behave similarly, but with a significant discontinuity since to the two halves of the domain have different properties in terms of control approach.

With the reference at $\theta_{ref} = \pm 180°$, a trained agent starting at $\theta_0 = 0°$ and at rest will have a policy guiding it to build up negative angular velocities in order to rotate in the right direction. Arriving at the reference, it can overshoot slightly, making the observation switch to $180°$ due to the normalization. After switching to the second half of the domain with a negative angular velocity, the policy will guide the agent to build positive angular velocities, overshooting again at the reference and looping back in a non-stationary fashion. In a setting with a circular boundary, a trained policy would be smooth at that boundary and make the agent progressively slow down, possibly at the cost of a steady-state error. However here, since the boundary is a discontinuity, the policy acts differently on either sides, influenced by the states visited during training on either halves of the domain. This idea can be verified by looking at the output of the critic network at $o_t^{-\pi}$ and o_t^{π}, the observations corresponding to the cases $\theta_t = -180°$ and $\theta_t = 180°$:

$$Q(o_t^{-\pi}, [u_\theta = 0, u_\phi]) > Q(o_t^{-\pi}, [u_\theta \neq 0, u_\phi]) \tag{2}$$

$$Q(o_t^{\pi}, [u_\theta = 0, u_\phi]) < Q(o_t^{\pi}, [u_\theta \neq 0, u_\phi]) \tag{3}$$

In a setting where the boundary was circular, the Q-value of slowing down ($u_\theta = 0$) would have been greater than the Q-value to act ($u_\theta \neq 0$) for o_t^{π}. We see that this is not the case, leading to a non-stationary behavior.

In most of the examples encountered in the literature, the goal is to stabilize a system around a certain fixed point. If proper work is done to recenter the coordinates, the discontinuity would not be a problem since the agent will learn to stay away from the discontinuity and can thus be left that way. In this work however, the goal is to be able to achieve reference tracking at any point. To get rid of the discontinuity, the observation has been defined to:

$$o_t = (\cos\theta, \sin\theta, \dot{\theta}, \cos\phi, \sin\phi, \dot{\phi}, \theta^{ref}, \phi^{ref}, \omega) \tag{4}$$

This definition comes at the cost of a larger observation space (9D instead of 7D), and thus harder convergence.

5 Reward Functions

The reward function is a key element of the RL problem formulation, as it embeds the agent's preferences and goals, which are necessary in order to guide the agent towards achieving a certain task. However, engineering a reward function is not a trivial task. For a certain class of problems, the reward function can be extracted easily from the context, such as the game score in some Atari games. In other problems, a careful balance between multiple conflicting attributes must be done in order to encapsulate all the challenges embedded in a given task. In the continuous control problem, this is particularly relevant and draws some of the same challenges as in designing the weighting matrices in the LQR objective function so as to balance a fast response with a realistic control law.

5.1 Quadratic Reward Function

Since the reward function indeed draws some of the same motivations as the LQR objective function, a logical first choice would be to specify a weighted squared error as reward function. We thus define the quadratic reward function (Q) $r_{t+1} = \rho(s_{t+1}, a_t) = \rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t)$ as a weighted quadratic function of the state vector $\boldsymbol{x}$ (see Chapter 4.1) and the control input $\boldsymbol{u}$:

$$\rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = -(\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1})^T \mathbf{Q} (\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1}) - \boldsymbol{u}_t^T \mathbf{P} \boldsymbol{u}_t$$

As a warm-up for this work, and to test the quadratic reward function, three configurations of the hyperparameters extracted from the baseline iterations found in [11] for DDPG, TD3 and SAC are evaluated using the quadratic reward function. These configurations feature different hidden sizes for more complex representations, different action noise standard deviations for exploration, different discount factors and learning rates. The hyperparameters can be found in the code and in [11]. The weighting matrices are tuned empirically. Plotting the multi-step return R_0 (called *Performance* here) against the number of epochs for all configurations yields the learning curves in Figure 6, and the statistics in Table 4.

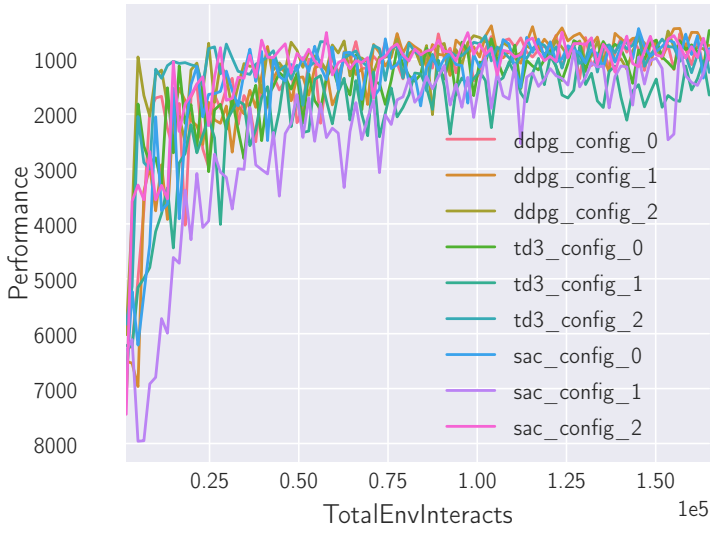

Figure 6: Learning curves for various configurations under the quadratic reward function

Table 4: Average reward for various configurations under the quadratic reward function

Mean Reward	Standard Deviation
-900	159

We see that all configurations, no matter the algorithm, converge after at most 1e5 environment interactions to an average performance of about -900 with a fairly narrow standard deviation, where an environment interaction is a single call to the `step` method. Such a performance is representative of a large steady-sate error. Since all configurations seem to fall rather quickly and systematically in the same local optima, the steady-state error is assumed to be caused by other factors than an algorithm's hyperparameter configuration, of which we will explore a few here.

Previous work, namely [18], has emphasized similar issues of the quadratic reward function. Those issues can be understood by comparing the RL and LQR objective functions:

- RL objective function (dropping the expectancy):

$$J_{RL} = \sum_{t=0}^{\infty} \gamma^t \rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t)$$

- LQR objective function:

$$J_{LQR} = \sum_{t=0}^{\infty} \rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t)$$

In a realistic setting, the infinite horizon discounted cumulative returns in RL are approximated by finite length trials:

$$J_{RL} \approx \sum_{t=0}^{N} \gamma^t \rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t)$$

Two main differences can be extracted from these approaches: the exponential discounting γ^t and the finite episode length in RL. Recall that the discount factor in RL is used to discount future rewards and to prove convergence of the algorithms.

A first approach would be to treat the finite length approximation, by increasing the episode length during training. We thus double the episode length and study the impact on performance. We will use a configuration of DDPG, picked from the different ones shown previously as a baseline, and test the doubled episode length using the same configuration with two different seeds, denoted with the superscript ^{*seed}. The baseline configuration is shown in Table 5.

Table 5: Configuration for quadratic reward environment tests

Parameter	Baseline
Epochs	100
Episode Length	110
Steps per Epoch	$110 \cdot 15$
Gamma	0.995
π Learning Rate	0.0025
Q Learning Rate	0.0025
Polyak	0.995
Batch Size	100
Start Steps	10000
Hidden Sizes	(400)
Actor Noise	0.1

The episodes will last longer, thus the total cumulative returns will be larger. Since we cannot use the cumulative returns as a performance score to compare both cases, we implement different metrics that are applied on 200 randomly generated initial conditions using the `reset` method, within the bounds defined. These metrics are:

- The Mean Absolute Error (MAE).
- The Mean Steady-State Error (MSSE).
- The percentage of trials ending in defined error margins.
- The percentage of trials that lead to unsteady behavior.
- The training time.
- The rise time.
- The settling time.
- The control input magnitude.
- The control input variation.

These metrics will be used throughout the rest of this work. However, all of the metrics will not be shown for each experiment, only the ones relevant to the experiment. For the metrics in Table 6, the error margin for qualifying for *in bounds* is set to ± 0.3 [rad] until further notice.

Table 6: Metrics for longer episode length under the quadratic reward function

Config.	Baseline	220 Ep. Length^{*0}	220 Ep. Length^{*10}
θ MAE [rad]	0.59	0.39	0.46
ϕ MAE [rad]	0.46	0.53	0.39
θ MSSE [rad]	0.55	0.34	0.37
ϕ MSSE [rad]	0.37	0.44	0.28
θ in bounds [%]	32.00	53.50	42.00
ϕ in bounds [%]	44.50	28.00	50.50
θ unsteady [%]	0.00	0.00	11.50
ϕ unsteady [%]	7.00	25.00	11.50
Convergence time (min)	14.70	30.84	31.47

From the lines highlighted in the table, we can see that the MSSE is reduced, in particular for the red gimbal. In general, the MAE is also reduced, indicating both a faster response and a smaller steady-state error. Yet, the improvement is not dramatic and the steady-state error is still very large with respect to control standards. A reason for this comes from the second difference that was mentioned earlier: the discount factor. In fact, the exponential discount will eventually cancel-off the cost of the steady-state-error in the limit $t \to \infty$, and thus render useless the efforts to increase the episode length, as also seen in [18].

In general, those issues emerge from the relative importance of the rewards at each time-step to the total cost for the quadratic reward function. Those issues will be further detailed and compared in Chapter 5.3. One way to deal with the cancelling of the steady-state error cost would be to increase the weights assigned in the weighting matrices. However even if the weights were tuned further, such an approach would not lead to generalizable results. Also, the slight improvement in performance also comes at the cost of a convergence time that is twice as long. In summary, such an approach is not encouraging and will not be pursued further.

By continuing to draw links with control theory, one can choose to augment the observation from the environment to the agent with integral terms i_θ and i_ϕ of the angular positions. In a control theory context, this approach stems from drawing links with integral terms added to correct static offsets. In a RL context, this approach stems from the fact that the Markov property may be difficult to satisfy without adding an extra memory term, which i_θ and i_ϕ emulate. The observation can thus be augmented:

$$o_t = (\cos\theta, \sin\theta, \dot{\theta}, \cos\phi, \sin\phi, \dot{\phi}, \theta^{ref}, \phi^{ref}, \omega, i_\theta, i_\phi)$$

Different forms of the integral term are experimented with:

- Integral:

$$i_\theta = \int (\theta^{ref} - \theta)$$

$$i_\phi = \int (\phi^{ref} - \phi)$$

- Scaled:

$$i_\theta = \frac{\int(\theta^{ref} - \theta)}{\max(\theta^{ref} - \theta) \cdot T}$$

$$i_\phi = \frac{\int(\phi^{ref} - \phi)}{\max(\phi^{ref} - \phi) \cdot T}$$

- Normalized:

$$i_\theta = \frac{\text{sign}\int(\theta^{ref} - \theta)}{1 + |\int(\theta^{ref} - \theta)|}$$

$$i_\phi = \frac{\text{sign}\int(\phi^{ref} - \phi)}{1 + |\int(\phi^{ref} - \phi)|}$$

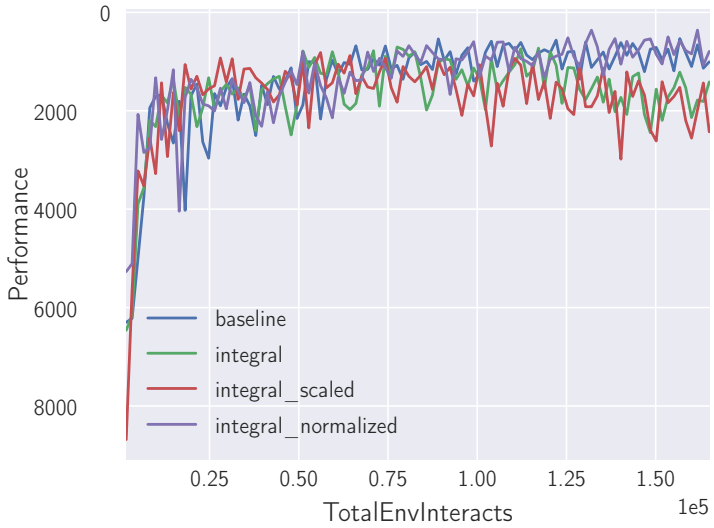

Figure 7: Learning curves for environments under the integral-augmented and quadratic reward functions

In figure 7, we can see a comparison of the learning curves with the baseline, from which we can directly compare the performance scores. The normalized variant of the integral follows a learning curve close to the baseline's, a hint that the agent does not exploit the additional observation, while the other two variants seem to showcase an unlearning phenomenon. An attempt at understanding the results for the normalized integral through a *state representation learning* analysis is presented in Section 8.

5.2 Variations of the Quadratic Reward Function

Since the different attempts at improving the performance using a quadratic reward function have not been successful, it is safe to assume that the issue comes from the reward function. Thus, we consider modifying the quadratic reward function in different ways in order to guide the agent towards the task that we wish to accomplish: zero steady-sate error. The different variations are:

- Quadratic with Ending Penalty (QEP):

$$\rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = -(\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1})^T \mathbf{Q} (\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1}) - \boldsymbol{u}_t^T \mathbf{P} \boldsymbol{u}_t - p$$

where:

$$p = \begin{cases} p, & \text{if ending error is out of bounds} \\ 0, & \text{otherwise} \end{cases}$$

- Quadratic with Penalty (QP):

$$\rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = -(\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1})^T \mathbf{Q} (\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1}) - \boldsymbol{u}_t^T \mathbf{P} \boldsymbol{u}_t - p$$

where:

$$p = \begin{cases} p, & \text{if current error is out of bounds} \\ 0, & \text{otherwise} \end{cases}$$

- Quadratic with Exponential (QE):

$$\begin{aligned} \rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = &- (\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1})^T \mathbf{Q} (\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1}) - \boldsymbol{u}_t^T \mathbf{P} \boldsymbol{u}_t \\ &- \alpha_\theta (1 - \exp\{-\beta_\theta (\theta^{ref} - \theta_{t+1})^2\}) \\ &- \alpha_\phi (1 - \exp\{-\beta_\phi (\phi^{ref} - \phi_{t+1})^2\}) \end{aligned}$$

- Quadratic with Bonus (QB):

$$\rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = -(\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1})^T \mathbf{Q} (\boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1}) - \boldsymbol{u}_t^T \mathbf{P} \boldsymbol{u}_t + b$$

where:

$$b = \begin{cases} b, & \text{if current error is within bounds} \\ 0, & \text{otherwise} \end{cases}$$

The idea behind the QEP and QB reward functions is to inform the agent of the performance it achieved at the end of the episode. The difference between the two is that a weak agent will be penalized systematically at the end of an episode by using QEP, while QB is a form of sparse reward. QP acts as a form of an integral term, by penalizing the agent throughout the episode for unsatisfactory behavior. QE, on the other hand, introduces a finer gradient near the goal in order to guide the agent towards the desired behavior. It is also a form of sparse reward as the active zone of the exponential is small, with a gentle gradient far from the goal inherited from the quadratic function. The different metrics are shown in Table 7.

Table 7: Metrics for variations of the quadratic reward function

Metric	Q	QEP	QP	QE	QB
θ MAE [rad]	0.59	0.77	0.52	0.40	0.63
ϕ MAE [rad]	0.46	0.54	0.37	0.29	0.46
θ MSSE [rad]	0.55	0.60	0.42	0.29	0.54
ϕ MSSE [rad]	0.37	0.48	0.31	0.21	0.40
θ in bounds [%]	32.00	40.00	38.00	65.50	28.00
ϕ in bounds [%]	44.50	34.50	46.50	74.00	27.50
θ unsteady [%]	0.00	0.00	0.00	0.00	5.00
ϕ unsteady [%]	7.00	0.00	0.00	3.00	8.00

We notice from the highlights that except from QE which improves on Q thanks to the added gradient near the goal, all variants fail to improve on the quadratic function. The reason stems from three factors: the same exponential discounting issue as seen in Chapter 5.1, the TD backup and the scale of the reward values.

The exponential discounting issue could be alleviated by tuning the penalty/bonus value, however the two other issues hinder any effort to do so. We will first explain the issue stemming from the TD backup operator, which concern mostly QB and QEP.

At the end of an episode, when receiving a reward with a penalty or a bonus, the TD backup that is used to calculate the loss in the DRL algorithms affects only the state-action pair (s_N, a_N) leading up to the episode termination. Thus, the penalty/bonus received will only affect this state-action pair, while the sequence of sub-optimal decisions that led up to a final steady-state error that is out of bounds are not impacted. If a similar sequence was to be repeated during training, the state-action pair (s_{N-1}, a_{N-1}) would now be impacted by the penalty/bonus of the previous sequence, and so on. We thus see that punctual rewards like these would take a large amount of iterations before flowing back and affecting the whole sequence of sub-optimal decisions taken.

Eligibility traces were introduced to alleviate such an issue. Roughly speaking, eligibility traces are decaying traces $e(s, a)$ of state-action pairs visited during an episode. A Q-value update of any state-action pair (s_t, a_t) is then performed on all $\{(s_0, a_0), \ldots, (s_{t-1}, a_{t-1})\}$ that led up to (s_t, a_t), multiplied by the decaying factor $e(s_i, a_i)$ that weighs the relevance of each (s_i, a_i) with respect to (s_t, a_t). However, because of how DRL algorithms are designed, namely the sampling in the replay buffer, eligibility traces are not trivial to implement and where thus not explored further. Active work is however done on such improvements, such as in [19].

Finally, scaling of the rewards might play an important role in the failure of these reward functions, most notably QP. It has been shown that having inputs or outputs of a DNN that lie at varying scales can hinder the learning process. The reason is that in this case, the DNN has to learn unbalanced distributions of weights and biases. Thus, it is common practice to scale the inputs, which we have done, and the outputs, which can lie at very different scales for QP depending on the response.

5.3 Alternative Reward Functions

Despite the efforts made, the quadratic reward function failed as a comprehensive expression of the control task. Thus, alternative reward functions were studied, with the hope that those would favor zero steady-state error solutions explicitly in their structure. Following the insights brought by [18], the absolute error reward function (A) was used. In fact, the absolute value is supposed to reduce the contribution of the initial, transient error, in favor of the steady-sate error. The squared-root error was evaluated in [18], but displayed unstable learning behaviors. A normalized (N) error function was also evaluated:

- Absolute (A):
$$\rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = -q^T \left| \boldsymbol{x}^{ref} - \boldsymbol{x}_{t+1} \right| - p^T |\boldsymbol{u}_t|$$

- Normalized (N):

$$\rho(\boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = -\frac{\frac{|\theta^d - \theta_{t+1}|}{k}}{1 + \frac{|\theta^d - \theta_{t+1}|}{k}} - \frac{\frac{|\phi^d - \phi_{t+1}|}{k}}{1 + \frac{|\phi^d - \phi_{t+1}|}{k}} - q_{\dot{\theta}}\dot{\theta}^2 - q_{\dot{\phi}}\dot{\phi}^2 - u_t^T \mathbf{P} u_t$$

The normalized reward function features a parameter k that controls it's slope. The response of the function is rather uniform if the absolute error is larger than k, while the gradient is steep if the error is smaller than k. Since the function is equal to $-\frac{1}{2}$ if the error is equal to k, this parameter represents, roughly speaking, the point at which we consider that the agent is achieving 50% of the desired task. This function is expanded with quadratic penalties on the velocities and control inputs. The structure of the function explicitly encourages exploration thanks to the uniform response far from the goal. Yet, the steep gradient helps convergence near the goal. It is therefore a form of sparse reward with a guiding gradient near the goal. Furthermore, the normalization solves the scaling issues discussed above. The performance of these two reward functions are presented in Table 8.

Table 8: Metrics for alternative reward functions

Metric	Q	A	N
θ MAE [rad]	0.59	0.41	0.49
ϕ MAE [rad]	0.46	0.34	0.32
θ MSSE [rad]	0.55	0.33	0.17
ϕ MSSE [rad]	0.37	0.25	0.16
θ in bounds [%]	32.00	51.50	82.50
ϕ in bounds [%]	44.50	54.50	89.00
θ unsteady [%]	0.00	0.00	0.00
ϕ unsteady [%]	7.00	0.00	0.00

The absolute reward function offers improvement comparable to QE, which is consistent with the results of [18]. The absolute value function reduces the importance of initial errors, while QE has a steeper slope near the goal. Those two different approaches both yielding improvements on Q, we can hypothesize that a function implementing both, such as N, may result in significant improvement. Looking at the results of N, we see that it offers a dramatic improvement on all previous reward functions on both the MSSE and the percentage of trials in bounds, as highlighted in Table 8.

As discussed previously, the normalized reward function explicitly encourages exploration. As a result, the agent is able to avoid local optimas that are associated with a fast transient response and large steady-state error. Consistent with the fact that N encourages exploration, convergence occurs in around 1.5e5 interactions, which is larger than for Q. With the other reward functions, such local optimas must be predominant in the reward landscape. As discussed, a reason why comes from the fact that the steady-state error cancels off, in favor of a large cost on the initial error. This can be studied by plotting the normalized returns contribution $\frac{R_{0:t}}{R_0}$ for an entire episode, as shown in Figure 8.

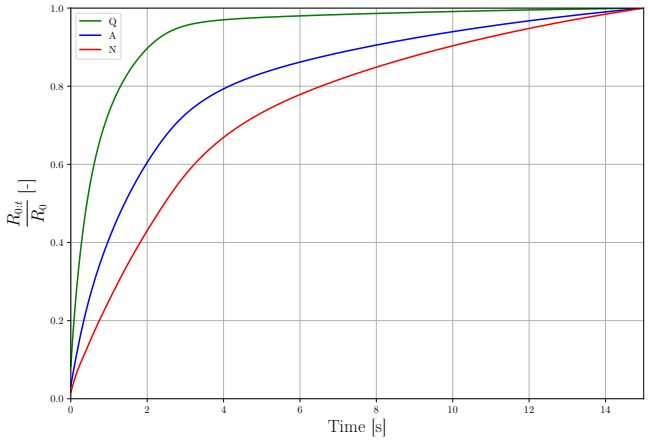

Figure 8: Comparison of the normalized discounted cumulative reward for three different reward functions

The results for Q and A are very similar as those found in [18]: for Q, the value of the total returns comes mostly from the contribution of the initial error, whereas A is able to reduce that effect, putting more emphasis on the steady-state error. An improvement on [18] comes from from the fact that the novel normalized reward function is able to decrease the effects of the initial error even more. For these reasons, the normalized reward function is chosen as basis for the rest of the work, while the others are not investigated further.

6 Simplified Model Training

Using the normalized reward function, an agent can be trained to learn a policy that achieves the control task on the gyroscope. In this section, friction is neglected in the model of the environment. We thus train agents on a *simplified* model of the gyroscope. The policy derived from such training may probably not be transferable to the control of a simulation with friction, even less for the true physical system; it nonetheless provides a good starting point. The *sim-to-real* issue, which is highly relevant when applying RL to robotics, will be tackled in Chapter 7.

6.1 Algorithm Choice

A single DRL algorithm will be chosen from the three different state-of-the-art algorithms discussed in Chapter 3 in order to focus the aim of the case study. Using the hyperparameter selection from Spinning Up's documentation and from [11], the configurations in Table 9 are used for comparison.

Table 9: Configurations of the 3 algorithms

Parameter	DDPG	TD3	SAC
Epochs	200	200	200
Episode Length	110	110	110
Steps per Epoch	$110 \cdot 15$	$110 \cdot 15$	$110 \cdot 15$
Gamma	0.995	0.995	0.995
π Learning Rate	0.001	0.0025	0.0005
Q Learning Rate	0.001	0.0025	0.0005
Polyak	0.995	0.995	0.995
Batch Size	100	100	100
Start Steps	20000	20000	20000
Hidden Sizes	(400)	(400)	(400)
Actor Noise	0.1	0.1	0.1
Policy Delay		2	
Target Noise		0.2	
Noise Clip		0.1	
Alpha			0.2

The performance evaluation for all three algorithms are presented in Table 10. In this table, the error margin for qualifying for *in bounds* is set to ± 0.05 [rad].

Table 10: Metrics for the 3 algorithms

Metric	DDPG	TD3	SAC
θ MAE [rad]	0.54	0.41	0.40
ϕ MAE [rad]	0.30	0.28	0.31
θ MSSE [rad]	0.36	0.22	0.15
ϕ MSSE [rad]	0.19	0.13	0.11
θ in bounds [%]	83.00	75.50	76.50
ϕ in bounds [%]	73.50	87.50	49.00
θ unsteady [%]	0.00	0.00	0.00
ϕ unsteady [%]	0.00	0.00	47.00
u_θ [V]	0.67	0.83	3.94
u_ϕ [V]	0.40	0.51	3.28
u_θ variation [V]	0.06	0.11	1.49
u_ϕ variation [V]	0.04	0.04	1.34

From the results highlighted in Table 10, it is clear that both TD3 and SAC offer better performances in terms of MSSE, that being especially true for SAC. It is important to note that these results come from a light search on the hyperparameters, and that a more extensive hyperparameter search may reveal different tendencies. Still, those results are coherent with the initial TD3 and SAC papers, in which those algorithms are shown to outperform DDPG on most tasks with less sensitivity to the chosen hyperparameters. However as it is highlighted at the end of Table 10, SAC showcases mean control voltages up to eight times as high as the ones from DDPG and TD3. The mean variation between subsequent control voltages, which is a measure of the speed of variation of the control law, can reach more than thirty-fold that of DDPG and TD3. The SAC agent is thus able to achieve better performance, at the cost of a policy that is much more erratic and unrealistic. Such a behavior can be controlled by tuning the entropy regularization. However, since SAC differs quite fairly from DDPG, the algorithm was not studied further to maintain the focus on DDPG and TD3. In that regard, TD3 was chosen as the base algorithm for the case study as it offers more conservative estimates of the Q-values and thus achieves better performance in general without diverging substantially from DDPG.

6.2 Reward Tuning

The reward function embeds the chosen trade-off between the different requirements that are associated to the task. For the normalized reward function, these are the error to the reference, the speed of the gimbals, and the magnitude of the control voltages. Before starting an extensive hyperparameter search, the parameters of the reward function are tuned in order to reflect the desired trade-off that we wish to obtain in the final control law. Namely, we wish for the response of the agent to be realistic to ease the transfer to the physical system, just like in LQR. As such, different configurations for the penalty on the control voltages are tested, as shown in Table 11.

Table 11: Reward configurations

Parameter	Cfg. n°0	Cfg. n°1	Cfg. n°2
k	0.05	0.05	0.05
$q_{\dot{\theta}}$	0.01	0.01	0.01
$q_{\dot{\phi}}$	0.01	0.01	0.01
$\mathbf{P}_{1,1}$	0.01	0.1	0.5
$\mathbf{P}_{2,2}$	0.01	0.1	0.5

The performance metrics associated with these configurations are presented in Table 12.

Table 12: Metrics for the reward configurations

Metric	Cfg. n°0	Cfg. n°1	Cfg. n°2
θ MAE [rad]	0.27	0.24	0.25
ϕ MAE [rad]	0.26	0.14	0.17
θ MSSE [rad]	0.06	0.07	0.08
ϕ MSSE [rad]	0.05	0.06	0.05
θ in bounds [%]	60.00	60.00	53.50
ϕ in bounds [%]	63.00	54.00	62.00
θ unsteady [%]	0.00	0.00	0.00
ϕ unsteady [%]	5.50	0.00	0.00
u_θ [V]	1.14	1.11	0.85
u_ϕ [V]	2.11	1.15	0.60
u_θ variation [V]	0.17	0.13	0.07
u_ϕ variation [V]	0.80	0.37	0.05

From the highlighted rows, it is clear that increasing the penalty on the control voltages yield control laws that are less varying in time and thus more realistic. This does not come at a significant cost in MAE and MSSE. We therefore choose the last configuration for the hyperparameter search.

6.3 Hyperparameter Search

The hyperparameter search method described here is not representative of an ideal machine learning workflow. In an ideal workflow, different configurations of hyperparameters would be tested with multiple seeds to avoid selecting configurations that performed well due to favorable random processes. Different hyperparameter tuning algorithms such as grid-search, random-search or even genetic algorithms would also be used in order to cover the hyperparameter domain. However, due to the speed bottleneck caused by the simulation, the absence of parallelization on the Spinning Up framework and to the short timescale of the case study, hyperparameter tuning was performed using a simplified algorithm suggested in [11] and with single seeds. The performance was evaluated using two seeds for the last iteration of the tuning process. It is important to note that since the performance is not measured simply by using the learning curves but by evaluating a set of metrics on randomly generated initial conditions, the problem of the seed is not as critical as in other cases.

The hyperparameter tuning algorithm from [11] is shown in Algorithm 1.

Algorithm 1: Hyperparameter tuning

Input: K hyperparameters, I baseline iterations;
for i = *1,2,...,I* **do**
  Train the agent with $H_i = \{h_{i,k}\}$;
  Evaluate the performance P_i;
  for k = *1,2,...,K* **do**
    Choose upper and lower bounds for hyperparameter $h_{i,k}$;
    Train the agent with $h_{i,k}^{up} = \textit{upper bound}$ and $h_{i,k}^{low} = \textit{lower bound}$;
    Evaluate the performances P_{up} and P_{low};
    if baseline performance P_i is beaten **then**
      $h_{i+1,k} \leftarrow \text{argmax}\{P_{up}, P_{low}\}$;
    end
  end
end

Such an algorithm is very intuitive in the sense that it drives the hyperparameter search in the direction of increasing performance for each hyperparameter. It performs $I(2K+1)$ experiments for I iterations and K hyperparameters, whereas grid-search would need $(2^K)I$ as noted by [11]. It is thus much more realistic considering the time and computational power constraints available. However, it does not cover the hyperparameter search domain well as it does not perform experiments on crossed combinations of hyperparameters, possibly falling short of better configurations.

Algorithm 1 is then applied to the simplified model with a TD3 agent and a tuned normalized reward function for $I = 4$. The configuration of hyperparameters at the end of each iteration are summarized in Table 13, and the corresponding learning curves and performance metrics in Figure 9 and Table 14 respectively.

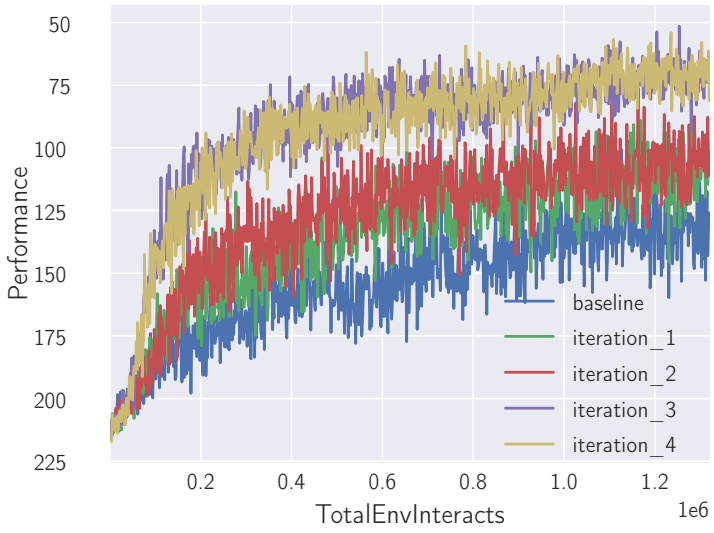

Figure 9: Learning curve for hyperparameter search

Table 13: Hyperparameter search iteration configurations

Parameter	Baseline	Iter. n°1	Iter. n°2	Iter. n°3	Iter. n°4
Epochs	800	800	800	800	800
Episode Length	110	110	110	110	110
Steps per Epoch	$110 \cdot 15$	$110 \cdot 15$	$110 \cdot 15$	$110 \cdot 15$	$110 \cdot 15$
Gamma	0.995	0.99	0.985	0.985	0.985
π Learning Rate	0.001	0.001	0.0015	0.0015	0.0015
Q Learning Rate	0.001	0.001	0.0015	0.0015	0.0015
Polyak	0.995	0.999	0.999	0.999	0.999
Batch Size	100	150	150	150	
Start Steps	20000	20000	20000	20000	20000
Hidden Sizes	(400)	(800)	(700)	(700,70)	(700,70,10)
Actor Noise	0.1	0.1	0.1	0.5	0.5
Policy Delay	2	2	2	2	2
Target Noise	0.2	0.2	0.2	0.2	0.2
Noise Clip	0.1	0.05	0.04	0.04	0.04

Table 14: Metrics for the hyperparameter search

Metric	Baseline	Iter. n°1	Iter. n°2	Iter. n°3	Iter. n°4^{*0}	Iter. n°4^{*10}
θ MAE [rad]	0.26	0.20	0.20	0.15	0.16	0.14
ϕ MAE [rad]	0.18	0.13	0.12	0.11	0.11	0.10
θ MSSE [rad]	0.07	0.05	0.02	0.0097	0.0060	0.0094
ϕ MSSE [rad]	0.05	0.02	0.02	0.0182	0.0134	0.0143
θ in bounds [%]	55.50	71.50	95.50	100.0	100.0	100.0
ϕ in bounds [%]	58.00	92.00	93.50	95.5	100.0	100.0
u_θ [V]	0.83	0.76	0.79	0.83	0.86	0.76
u_ϕ [V]	0.65	0.58	0.62	0.52	0.49	0.47
u_θ variation [V]	0.07	0.06	0.07	0.07	0.08	0.07
u_ϕ variation [V]	0.05	0.07	0.08	0.06	0.07	0.06

To analyze the results of this hyperparameter search, the major takeaway of each iteration is studied:

- Iteration n°1: the learning curve shows convergence to a performance score of nearly -100 compared to -125 for the baseline, and the slope of the learning curve in the first 1e5 interactions is much steeper. The MSSE is roughly decreased by a factor of two while the percentage of trials in bounds is drastically improved. Most probably, a greater number of neurons in the hidden layer is necessary in order to build a regressor that is complex enough to approximate the Q-function and the policy.

- Iteration n°2: the slope of the learning curve is not improved, however it shows convergence to a performance score that is slightly greater. In fact, the MSSE is cut in half for the red gimbal, and the percentage of trials in bounds is close to 100%. A hypothesis would be that the bigger learning curve pushes the network to avoid previous local optimas. Since the increase in performance is slight, we decide to investigate deeper networks and more action noise for the next iterations.

- Iteration n°3: the slope of the learning curve in the first 1e5 interactions is drastically steeper, and shows convergence at a much higher score, namely -70. The MSSE is cut in half again for the red gimbal with a slight improvement for the blue gimbal, while the percentage of trials in bounds is 100% for the red gimbal. Two major hyperparameter changes can explain those results: adding an extra hidden layer allows the function approximator to decouple the feature extraction in two stages and is able to learn a vastly more complex approximation of the Q-function and policy, allowing the score to be much higher. The steepness of the learning curve slope can be explained by the increased actor noise that helps the agent explore more in the early stages and avoid falling in critical local optimas.

- Iteration n°4: the learning curve does not change. The MSSE is slightly improved, particularly for seed 0, while the percentage of trials in bounds as now reached 100% for both gimbals. The extra layer may help the network decouple the processing stages even more and thus achieve the 100% in bounds, however further hyperparameter search would be necessary to assess the validity of that conclusion.

The agent trained at iteration n°4 using seed 0 will be used as the baseline for any further comparative analysis.

6.4 Response Shaping

A further analysis of the effects of the choice of weights in the reward function is conducted by training an agent with the hyperparameter configurations of iteration n°4 with a normalized reward function defined by the weights in Table 15. Configuration n°0 and n°1 respectively apply a decreasing and increasing weight on the control voltages penalty, while configuration n°2 and n°3 apply an increasing weight on the gimbal velocities. The corresponding performance metrics are presented in Table 16.

Table 15: Reward configurations for response shaping

Parameter	Baseline	Cfg. n°0	Cfg. n°1	Cfg. n°2	Cfg. n°3
k	0.05	0.05	0.05	0.05	0.05
$q_{\dot{\theta}}$	0.01	0.01	0.01	0.1	1
$q_{\dot{\phi}}$	0.01	0.01	0.01	0.1	1
$\mathbf{P}_{1,1}$	0.5	0.1	1	0.5	0.5
$\mathbf{P}_{2,2}$	0.5	0.1	1	0.5	0.5

Table 16: Metrics for the response shaping reward variations

Metrics	Baseline	Cfg. n°0	Cfg. n°1	Cfg. n°2	Cfg. n°3
θ rise time (s)	0.44	0.50	0.52	0.71	/
ϕ rise time (s)	0.34	0.33	0.52	0.65	/
θ settling time (s)	0.72	0.76	0.92	1.28	/
ϕ settling time (s)	0.65	0.56	0.87	0.96	/
u_θ (V)	0.86	0.99	0.71	0.63	/
u_ϕ (V)	0.49	0.57	0.47	0.52	/
u_θ variation (V)	0.08	0.09	0.06	0.06	/
u_ϕ variation (V)	0.07	0.08	0.05	0.04	/

Table 16 clearly shows the effect of adjusting the trade-off expressed in the reward function. Decreasing the penalty on the control voltages makes the control law more erratic, but without a significant effect on the rise and settling time, most probably because the response are already sufficiently optimized in that regards. Increasing the penalty on the control voltages makes the control law more realistic, at the cost of a slower overall response. Increasing the penalty on the gimbal velocities nearly doubles the rise and settling times. In configuration n°3, the penalty on the velocities is such that the algorithm is not even able to converge. Reward tuning is thus important to assure convergence, realistic control laws and to shape the overall response.

Table 17: Metrics for the RL agent and the FL controller

Config.	Baseline	FL Controller
θ MAE [rad]	0.16	0.2434
ϕ MAE [rad]	0.11	0.2931
θ MSSE [rad]	0.0060	0
ϕ MSSE [rad]	0.0134	0
θ rise time [s]	0.59	1.03
ϕ rise time [s]	0.45	0.99
θ settling time [s]	0.79	1.79
ϕ settling time [s]	0.73	1.90
u_θ [V]	0.86	1.32
u_ϕ [V]	0.49	1.05
u_θ variation [V]	0.08	0.12
u_ϕ variation [V]	0.07	0.07

6.5 Control Evaluation

As a comparative analysis between the FL controller and the RL agent trained in a simplified model of the gyroscope, performance metrics are presented in Table 17.

Table 17 highlights the main benefits of each approach. Namely, the FL controller manages to reach a MSSE of 0, while the baseline RL agent still produces a MSSE of an order of magnitude of roughly 1e$-$2rad. On the other hand, the RL agent is able to cut the rise and settling time of the FL controller by half, with even smoother control voltages.

Visual analysis of the performances of the performance of the RL agent is performed in Figure 10, 11 and 12 with respectively step tracking, multi-step tracking and sine tracking experiments. These experiments showcase the fast response of the RL agent.

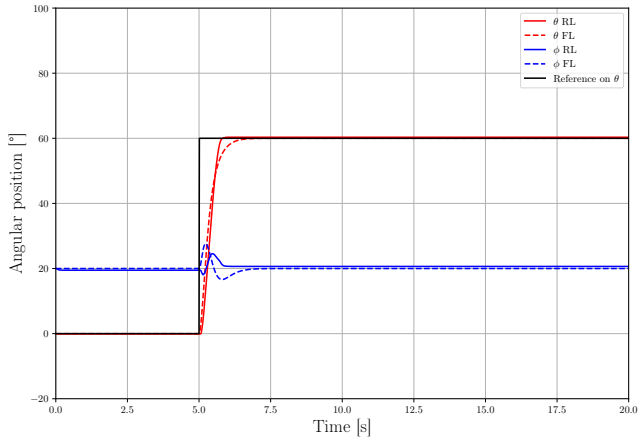

Figure 10: Step tracking comparison between the RL agent and the FL controller for $\theta_{ref} = 60°$ and $[\theta_0, \phi_0] = [0°, 20°]$ at 200 RPM

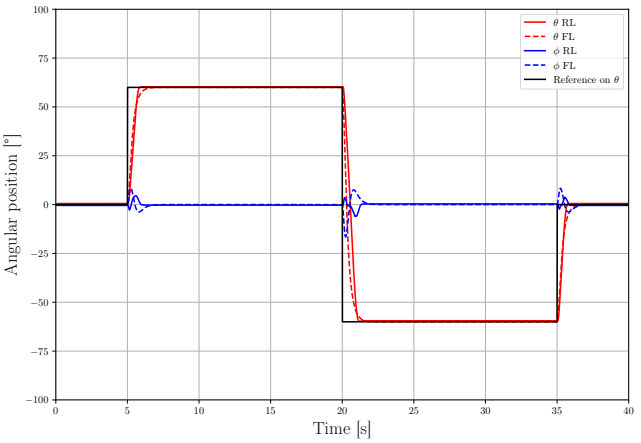

Figure 11: Multi-step tracking comparison between the RL agent and the FL controller for $\theta_{ref} = 60°, -60°$ and $[\theta_0, \phi_0] = [0°, 0°]$ at 200 RPM

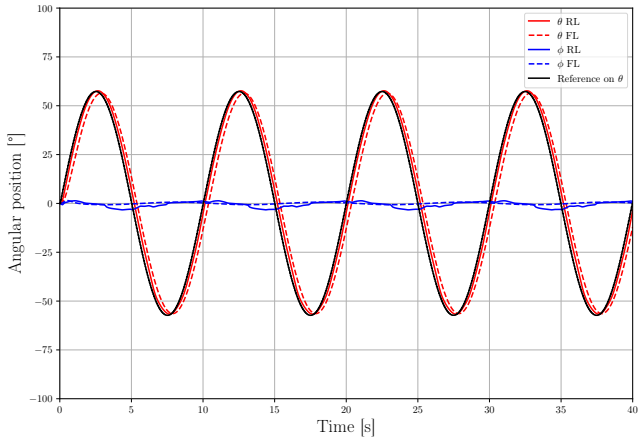

Figure 12: Sine tracking comparison between the RL agent and the FL controller on θ for A = 60°, f = 0.1 Hz and $[\theta_0, \phi_0] = [0°, 0°]$ at 200 RPM

7 Domain Adaptation

The agent trained in the previous section learned a policy suited for a simplified and frictionless model of the gyroscope. Whether the agent will be able to perform well on a simulation with friction, or most importantly on the true physical system, is questionable. A study of the sim-to-real problem, namely the design of methods to learn policies that can achieve the desired goal on the physical system, is thus required to obtain a policy that is relevant in the robotics context of this case study.

As expressed by [20], transferring control policies from a simulation to their physical counterpart is a form of domain adaptation. Domain adaptation consists of transferring a model trained in a source domain (here the simulation), onto a target domain (here the physical system). Such an approach works based on the assumption that the two domains share relevant characteristics that can be exploited in both.

7.1 Randomization of Dynamics

There exists multiple methods to execute domain adaptation in the context of RL for robotics. An approach would be to perform *transfer learning* by fine-tuning the pre-trained model on the true physical system, that way the agent can refine it's policy and adapt to the specific dynamics of the system. That way, the problem of the time constraints of training the model directly on the true system is alleviated, as the model learns the high-level control characteristics in simulation, and the finer ones on the true system. However such an approach still suffers from the security issues of setting up such an experiment, and due to the COVID-19 outbreak, this approach was discarded.

Instead, this transfer learning approach can be done on a simulation of higher fidelity. For instance, Coulomb and viscous friction can be added to the model, with constants identified through optimization on physical data. Transfer learning can then be performed on a simulation based on such a model. However, as noted by [5], system identification on the friction constants yields different results for different experiments, a sign that the Coulomb and viscous friction model used are not representative of the true physical system. As pointed out earlier, building a more complex model of the system would make a RL approach irrelevant. A robust alternative to training a model on a simulation with identified friction constants, is to train the model on a simulation with randomized friction constants. Such an approach, called *domain randomization*, models differences between the source and target domains as variability in the source domain. The agent is forced to learn a policy that is robust to the uncertainties in the dynamics of the system. In addition, noise can be added to the constants of inertia to achieve this goal.

Formally, the framework is similar to the one employed by [17] and [20]. Namely, the goal is to train a policy that can control the gyroscope under the dynamics of the real world $P^*(s'|s,a)$. For practical purposes, the policy is instead trained on estimated dynamics $\hat{P}(s'|s,a) \approx P^*(s'|s,a)$ in a simulation. For robustness, the policy is trained on randomized dynamics models, where a set of dynamics parameters is parametrized by λ. The dynamics of the simulation is then $\hat{P}(s'|s,a,\lambda)$ and the objective function subject to the distribution ρ_λ of dynamics models becomes:

$$J(\pi, \lambda) = \mathop{\mathbb{E}}_{\lambda \sim \rho_\lambda} \left[\mathop{\mathbb{E}}_{\tau \sim \pi, \lambda} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi, \lambda \right] \right]$$

The Gym environment is thus set-up with a randomization method that draws a new set of inertia and friction constants at each environment reset:

- The Coulomb and viscous friction constants are drawn from a uniform distribution in the range $[0, 0.05]$.
- Gaussian noise is added to the inertia constants with standard deviation equal to 30% of the default value.

7.2 Adaptability Evaluation

For comparison before training a robust policy, the FL controller and the baseline policy trained in the previous section are tested on the robust environment to evaluate their performance on unseen dynamics. The previous step tracking experiment is thus reiterated on the new environment for both the FL controller and the RL agent, respectively in Figure 13 and Figure 14. In both figures, the results are shown for both the simplified environment and the robust environment. Let us recall that the friction constants used to derive the FL controller are the ones from Table 3. For the experiment however, the friction constants of the environment were set to the ones in Table 18, while the inertias were set to 125% of their default value.

Table 18: Friction constants

f_{cr}	f_{vr}	f_{cb}	f_{vb}
0.001	0.002679	0.001	0.005308

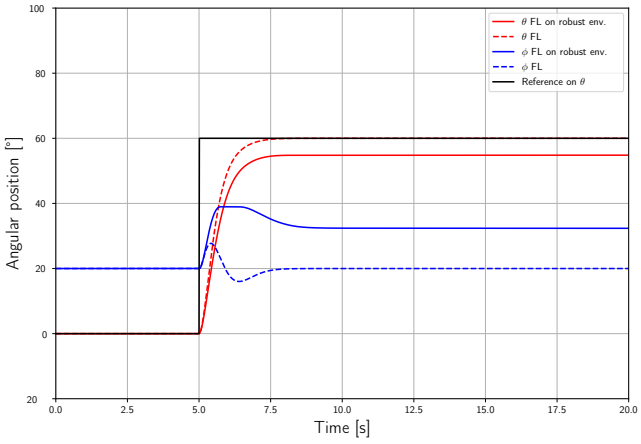

Figure 13: Step tracking comparison for the FL controller in the classic and robust environments for $\theta_{ref} = 60°$ and $[\theta_0, \phi_0] = [0°, 20°]$ at 200 RPM

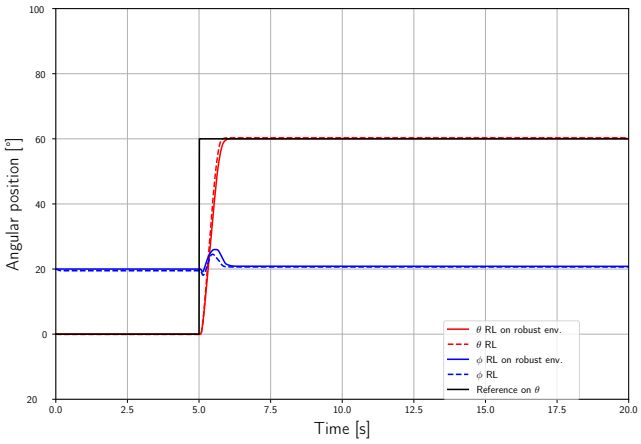

Figure 14: Step tracking comparison for the RL agent in the classic and robust environments for $\theta_{ref} = 60°$ and $[\theta_0, \phi_0] = [0°, 20°]$ at 200 RPM

Figure 13 shows clearly that the FL controller is not able to generalize to variability in the dynamics of the environment and is highly dependent on the accuracy of the model. It has to be noted that this issue can be alleviated by adding an integrator, as done in the last part of [5]. In comparison, Figure 14 shows that the baseline agent performs just as well in the robust environment as in the simplified environment.

To put this surprising last result into perspective, domain adaptation is performed by train-

ing the same agent (iteration n°4 of Table 13) on the robust environment. The performance metrics of the baseline and the robust agents for 200 random resets of the dynamics and initial conditions of the gyroscope are presented in Table 19.

Table 19: Metrics for agents on the randomized environment

Config.	Baseline	Robust
θ MAE [rad]	0.15	0.16
ϕ MAE [rad]	0.13	0.09
θ MSSE [rad]	0.0072	0.0071
ϕ MSSE [rad]	0.0139	0.0131
θ in bounds [%]	100.00	100.00
ϕ in bounds [%]	98.50	99.50
u_θ [V]	1.0143	0.9551
u_ϕ [V]	0.8122	0.7348
u_θ variation [V]	0.21	0.14
u_ϕ variation [V]	0.20	0.11

As highlighted in Table 19, it can indeed be seen that the MSSE metrics are similar for both agents. It could be concluded that the domain adaptation is not needed and that the baseline policy is already robust to variability in the dynamics of the environment. The last two highlights however, namely the control voltage variation metrics, show that the baseline agent acts more erratically when faced with unseen dynamics whereas the robust agent is able to learn a policy that is more realistic and thus that might generalize better to the dynamics of the real system.

The domain adaptation work presented above is targeted at variability in the dynamics of the gyroscope, namely the inertias and the friction parameters. Another source of adaptability can be studied: variability in the disk rotation speed ω. Such a variability is considerably different. In fact, whereas the dynamics of the environment are not readily accessible by the agent, the disk rotation speed is, as it is measured in the physical system and part of the observation o_t fed to the agent. The agent is thus robust to this parameter without an apparent need for domain adaptation. However, the disk angular velocities are varied from an episode to the next, but kept constant throughout an episode. A study can therefore be made on the adaptability of the agent to disk angular velocities that vary throughout an episode. A step tracking experiment is therefore set up with a disk rotation speed following a sine of mean 200RPM, amplitude 10RPM, and varying frequency.

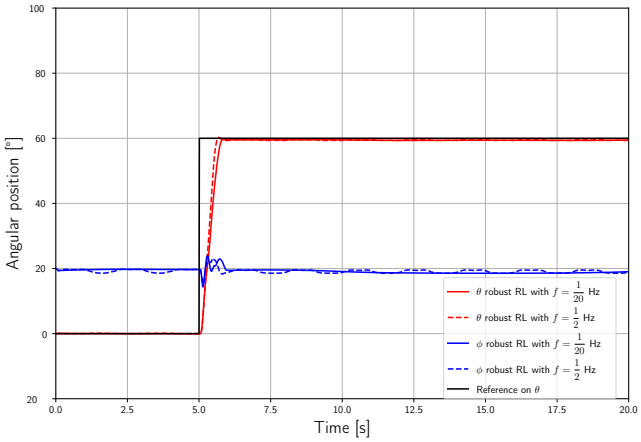

Figure 15: Step tracking for the robust RL agent in the robust environment for $\theta_{ref} = 60°$ and $[\theta_0, \phi_0] = [0°, 20°]$ with a disk angular velocity following a sine wave centered at 200RPM of amplitude 10RPM and varying frequency

Results in Figure 15 show that the robust agent is able to generalize and control the gyroscope to a satisfactory level. However, it is possible to observe non-stationary behaviors in the blue gimbal that depend on the disk rotation speed frequency. Such a behavior can be explained by the fact that the agent was trained with (observed) disk angular velocities that were constant throughout an episode. In further work, training the agent with disk angular velocities that vary throughout an episode could be investigated, in the same fashion as it was done for the parameters of the dynamics model.

8 Interpretability

Leveraging DNNs as function approximators has considerably transformed the field of RL. The success of DRL algorithms, and of DNNs in general, comes from their ability to automatically learn high-level representations of the data in order to execute a desired task, be it classification or regression. However this makes DNNs hard to understand. Interpretability in AI is a growing research interest [21]. It is also highly relevant for DRL as understanding what the agent has learned is key in order to improve the agent's performance in the design loop, as seen in the famous DQN paper [22].

8.1 Representation Learning

In RL, observations that are spatially similar can result in very dissimilar control rules. This is why DNNs are powerful function approximators for RL as they can automatically learn highly non-linear feature representations of the observation. Each layer of the DNN can be viewed as consisting of an intermediate stage of the representation, the final representation being used and combined at the output of the critic function approximator to compute the optimal action to be taken. The Internal Model Principle (IMP) [23] of control theory informally states that a controller must contain a subsystem tasked with predicting, in some form, the dynamics of the system. In that regards, the feature representation learned by the DNN must allow it to easily discriminate between observations that differ from a dynamical perspective [24].

8.2 Dimensionality Reduction with t-SNE

In order to highlight this aggregation (respectively discrimination) between dynamically similar (respectively dissimilar) observations, the activations of each hidden layer of a DNN can be studied. To this end, the t-SNE algorithm [25] is used to lower the dimensionality of the activations. The t-SNE algorithm is a dimensionality reduction technique that can capture aggregation in the data at different scales.

To do so, t-SNE starts by converting the high-dimensional Euclidean distances between datapoints into pairwise conditional probabilities. The similarity of datapoint x_j to x_i is thus defined as a conditional probability $p_{j|i}$ that is proportionate to a Gaussian centered at x_i:

$$p_{j|i} = \frac{\exp\{-||x_i - x_j||^2/2\sigma_i^2\}}{\sum_{k \neq i} \exp\{-||x_i - x_k||^2/2\sigma_i^2\}}$$

where σ_i, the variance of the Gaussian at x_i, is computed to be small for datapoints in dense regions and large for sparser regions through a hyperparameter called *perplexity*, which controls the number of neighbors to any datapoint. The joint probabilities are then computed:

$$p_{ij} = \frac{p_{j|i} + p_i|j}{2N}$$

Using this computed pairwise probability distribution in the high dimensional space, the

goal is to preserve this relationship in the lower dimensional space. To do so, the joint probabilities between y_i and y_j in the lower dimensional space are expressed using a Student t-distribution, which has a better tendency to reduce crowing in lower dimensions:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq i}(1 + ||y_i - y_k||^2)^{-1}}$$

In order to locate the datapoints y_i on the newly created lower dimensional map, gradient descent on the Kullback-Leibler divergence of the two distributions is done an the optimization loop. This way, the probability distribution in the two spaces will be made similar.

It is important to note that t-SNE works on a defined dataset, thus the dimensionality reduction cannot be applied to new testing datapoints. It is thus purely a visualization technique.

8.3 Visual Analysis of Learned Representations

To apply the t-SNE algorithm, 1000 observations are randomly generated with a set of constraints defined by the experiment. For each observation, the activations of the actor's hidden layers are stored, as well as the control inputs of the actor network. The activation data, where each datapoint has dimensionality N_i corresponding to the number of neurons of layer i, is then fed to the t-SNE algorithm for a 2D dimensionality reduction. To visualize the clusters results from t-SNE, each datapoint is colored according to the actor network output (which is the control input) for the observation that generated that datapoint, as well as attributes of the observation such as the gimbal and disk angular velocities. This coloring is so that clusters can be identified with respect to dynamical similarities of the input observations. Finally, the same procedure is done with a DNN with random weights as a control to put the results obtained into perspective.

As highlighted in Chapter 5.1, for the observation augmented with a normalized integral, the similarities with the baseline learning curve suggest that the DNN learns to discard the extra observation. To verify such a hypothesis, t-SNE can be applied to visualize the activations of the single hidden layer of the DDPG baseline configuration of Chapter 5.1. To make the cluster identification easier, the observations are generated with a set of constraints:

- ϕ, $\dot{\phi}$, ϕ_{ref} and i_ϕ are set to 0 in order to focus the identification on θ.
- θ is set to 0° while θ_{ref} is set to 5° in order to study the steady-state error case.

The t-SNE results are colorized according to three parameters: the actor network control inputs u_θ, the angular velocity $\dot{\theta}$ and the normalized integral i_θ of the red gimbal. The results are presented for a trained agent in Figure 16 and for a control with the same configuration in Figure 17.

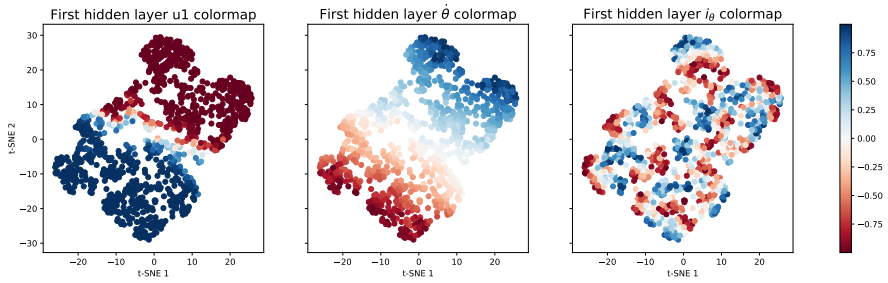

Figure 16: Colorized t-SNE results for the integral-augmented agent with u1$\equiv u_\theta$

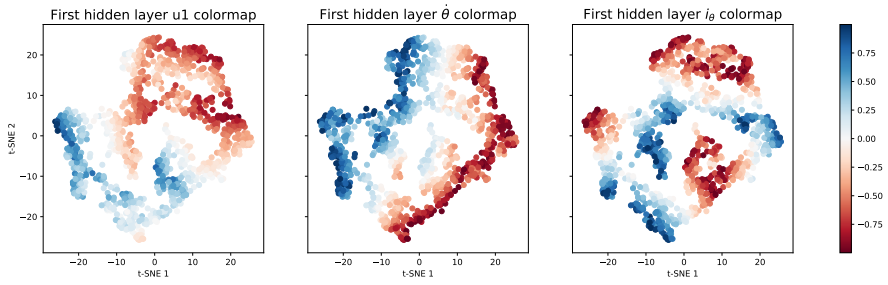

Figure 17: Colorized t-SNE results for the integral-augmented control with u1$\equiv u_\theta$

In Figure 16, the $\dot{\theta}$ colormap shows that the activations for similar velocities are clustered together. By looking at the corresponding control, it could be said that the clustering is not due to the network learning anything since the clustering looks similar for the control. However, by looking at the u_θ colormap, it can be seen that positive velocities correspond to negative control inputs and vice-versa. This is intuitive as a positive velocity means that the gimbal is turning in the right direction and that it needs to be slowed down to avoid overshoot, hence the negative voltage. In the control, such a correspondence cannot be seen. This analysis suggests that the DNN's hidden layer is able to learn a representation that discriminates between angular velocities, which is used at the final output to compute the proper control. However such a correspondence cannot be seen in the i_θ colormap, which looks random. It can thus be assumed that the DNN learns to discard the normalized integral observations.

Similarly, this study can be conducted for the robust agent of Chapter 7.2 to study the activations at each of the three hidden layers. To make the cluster identification easier, the observations are generated with a set of constraints:

- ϕ, $\dot{\phi}$ and ϕ_{ref} are set to 0 in order to focus the identification on θ.
- θ is set to 0° while θ_{ref} is set to 60° in order to study the step-tracking case.

The t-SNE results are colorized according to three parameters: the actor network control input u_θ and the angular velocity $\dot{\theta}$ of the red gimbal, as well as the disk's angular velocity

ω. The results are presented for a trained agent in Figure 18 and for a control with the same configuration in Figure 19.

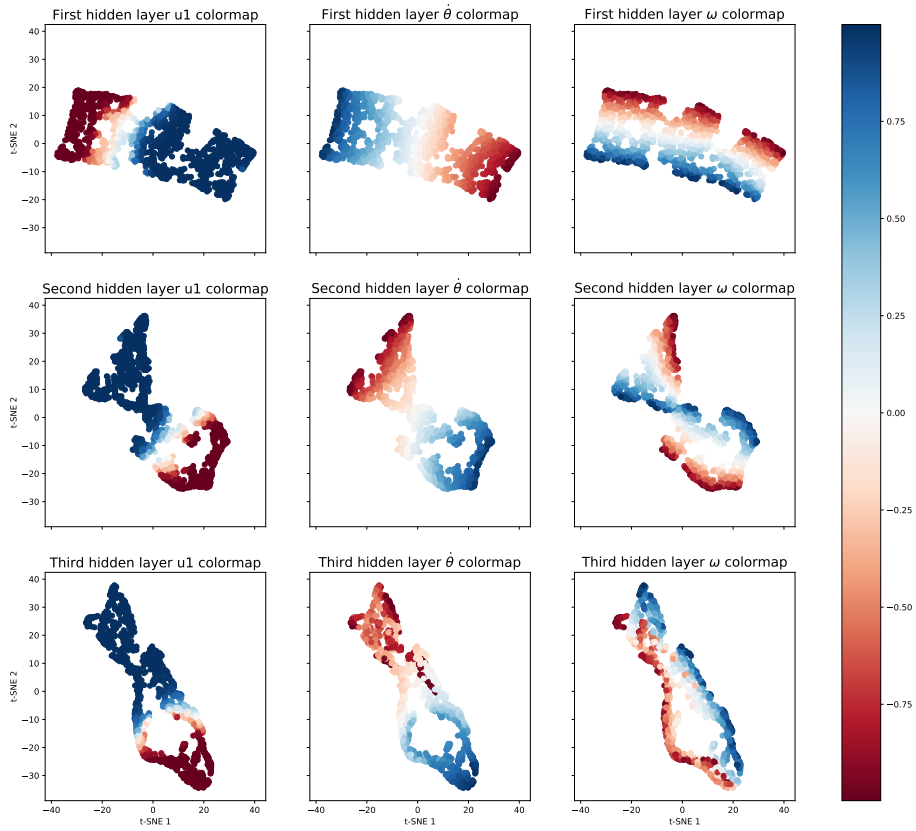

Figure 18: Colorized t-SNE results for the robust agent with u1$\equiv u_\theta$

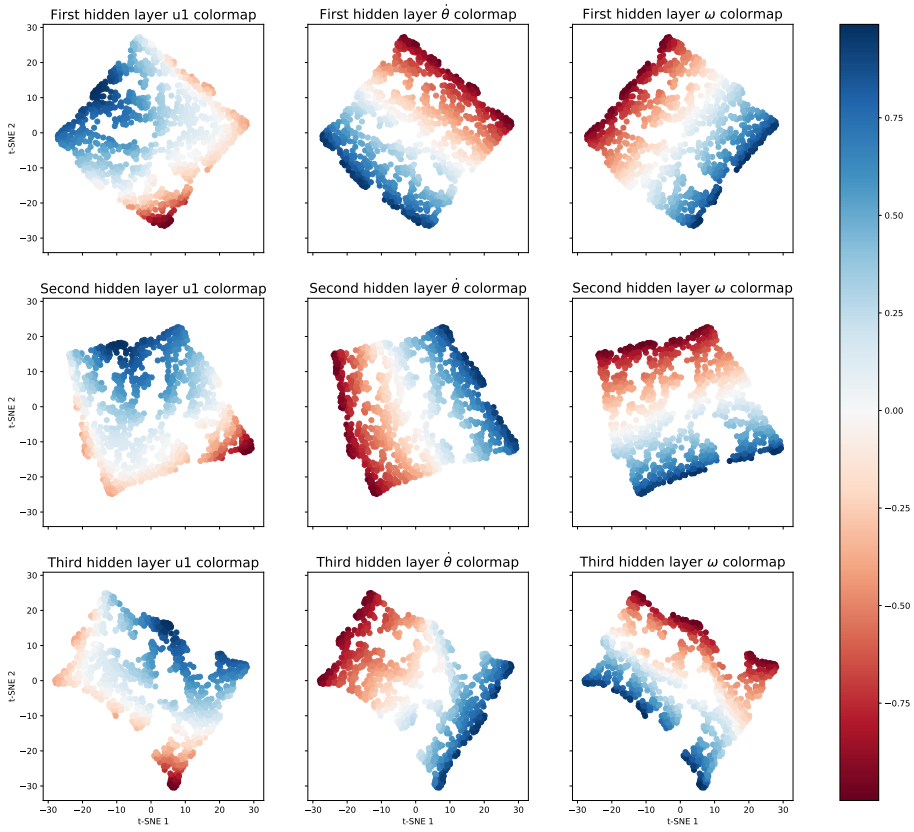

Figure 19: Colorized t-SNE results for the robust control with u1$\equiv u_\theta$

Again, the same correspondence between velocity and control input can be seen from the u_θ and $\dot{\theta}$ colormaps, a clue that the DNN is able to learn a representation that discriminates between angular velocities. As the clustering gets finer after each layer, it can be assumed that the representation gets more abstract and general deeper into the network. It is important to note at this point the importance of comparing with the control. We see that t-SNE also groups datapoints by color for the control. However, this grouping shows none of the correspondence highlighted for the trained agent. Furthermore, the color gradient is mostly linear and smooth, a sign that the input data is not manipulated throughout the layers. With these considerations and by looking at the ω colormap, it is not safe to say that the network is able to learn a representation that discriminates for the disk's angular velocity before the third hidden layer. In fact, for the first two hidden layers, the visualization are comparable to the control. In the last hidden layer however, there is a clear distinction between negative velocities on the left and positive ones on the right, with low velocities

(positive or negative) fused in the middle. There is no correspondence with the u_θ colormap however, it can thus be assumed that this information is not relevant for this experiment as it is not used at the output. Since the representation has been learned, it can be assumed that it is relevant for other settings. Further investigation can be conducted using the workflow presented here on other carefully chosen experiments in order to highlight other features of the representation and to better understand the policy learned.

9 Going Further

The results derived throughout this work have shown that DRL algorithms can learn powerful policies that can adapt to uncertainty in the dynamics of the model in an end-to-end fashion that requires little domain knowledge and rather simple dynamical models. However the careful design loop has not been able to permanently cancel the steady-state error, which makes the derived agent inferior in terms of control standards. This realization paves the way for further investigations, most notably in the areas stated below:

- Implementation: the Spinning Up framework has been used extensively throughout this case study, as it offered easily accessible resources which were perfect to start working fast. However, the absence of parallelization and the lack of scheduling for noise and learning rates has been ultimately detrimental. Parallelized algorithms would allow for much more tests to be done, which would result in a more rigorous hyperparameter search that could unlock even better agents. Thus, it is highly recommended to use RLlib from the start, or other similar frameworks. In addition, the failure of the integral-augmented observation should be studied in more details as to identify the cause of the unlearning phenomenon. Other ways of adding memory to counteract the steady-state error should be investigated.

- Inverse RL: by looking at the difficulty posed by reward engineering for the gyroscope, it is natural to consider automatic reward design paradigms, such as Inverse Reinforcement Learing (IRL). The goal of IRL is to learn a reward function that best explains a set of expert demonstrations. In that way expert demonstrations could be generated using existing controllers, such as the FL controller, and the reward function could be learned from those. Maximum Entropy IRL does not assume that the expert follows the optimal policy, as such the reward function can still be learned from sub-optimal demonstrations [26]. IRL is a form of imitation learning, which does not seek the reward explicitly, but could also be studied as a way to leverage existing controller.

- System identification: the randomization of dynamics method presented in Chapter 7.1 has shown promising results. However it relies only on a transfer learning approach to increase robustness to variability in the dynamics. An alternative would be to add a system identification module to the agent that uses past states and action in order to predict the dynamics parameters $\hat{\mu}$. A policy could then be trained using this module: $\pi(a_t|s_t, \hat{\mu})$. In [20], a recurrent architecture is used to fuse this identification module directly onto the actor-critic architecture.

- Interpretability: the analysis of the learned representation was only an overview of the possibilities at hand to understand the policy learned by the agent. More work can go into exploring t-SNE for other configurations. Furthermore, [21] presents a framework to automatically analyze the policy for games, which could be applied to the context of this case study.

10 Conclusion

The aim of the case study was to analyze the feasibility of using DRL to train agents capable of learning to control a gyroscope. While doing so, the DRL workflow for robotics was challenged and refined in order to build a consistent way of approaching such problems. In the end, the trained agent was capable of reaching a satisfying control performance. Furthermore, it proved to be robust to uncertainties in the dynamics parameters. The steps required to achieve this include: building a simulation environment of the gyroscope compatible with standard RL frameworks, a careful analysis of reward functions, hyperparameter search aimed at reducing the steady-state error, and domain adaptation using randomized dynamics to increase robustness. The trained agent was then analyzed using t-SNE to better understand the policy derived.

Building an environment is key in any RL problem. As demonstrated, naive definitions of the observation returned by the environment can lead to unintuitive results. As such, changing the definition of the observation to circumvent the discontinuity issue was essential in order to learn a proper policy at any desired tracking point.

The issue of drawing too strict parallels with control theory also emerged when choosing the quadratic function as a reward signal. After analysis it became apparent that better choices of reward function exist, which encourage more the agent to get rid of the steady-state error. However, a systematic solution was not found to get completely rid of this issue, and IRL could be studied as an alternative to manual reward engineering. Also, the unlearning phenomenons observed for the integral-augmented observation still remain unanswered. Analysis using interpretability techniques could be performed in order to better understand this behavior.

The hyperparameter search helped in dramatically reducing the steady-state error, most notably by building deeper networks and with more exploration incentives. In the end, the trained agent outperformed the FL controller in terms of speed of response, but not in steady-state error.

A simple domain adaptation technique showed that an agent robust to uncertainty in dynamics parameters could be trained without the need for high fidelity simulators. This result builds confidence in the ability of the agent to be deployable on the true system.

The main shortcoming of this work remains the absence of any validation on the real system. With that in mind, it remained a driving guideline of the project to be able to question the true capabilities of the agent (through domain adaptation and interpretability analysis), as it has been observed that RL agents can exploit idiosyncrasies of the simulator. Thus, validation and testing on the true system is an essential improvement.

References

[1] K. Fragkiadaki *Deep reinforcement learning and control*. Carnegie Mellon, School of Computer Science, CMU 10703 Lecture Series.

[2] D. Silver et al. *Mastering the game of Go with deep neural networks and tree search*. Nature 529, 484–489, 2016.

[3] A. Puigdomènech Badia et al. *Agent57: outperforming the Atari human benchmark*. arXiv preprint, arXiv:2003.13350, 2020.

[4] Quanser *3 DOF Gyroscope*. URL: `https://www.quanser.com/products/3-dof-gyroscope/`.

[5] Y. Agram *Identification and Control of a Gyroscope*. Semester Project, EPFL Automatic Control Laboratory, 2018.

[6] T. Dghaily *Nonlinear Identification and Control of a Gyroscope using Neural Networks*. Semester Project, EPFL Automatic Control Laboratory, 2019.

[7] H. Li, S. Yang and H. Rem *Dynamic decoupling control of DGCMG gimbal system via state feedback linearization*. Mechatronics, 36:127–35, 2016.

[8] J. Montoya–Cháirez, V. Santibáñez and J. Moreno–Valenzuela *Adaptive control schemes applied to a control moment gyroscope of 2 degrees of freedom*. Mechatronics, 57:73–85, 2019.

[9] Open AI *Spinning Up*. URL: `https://spinningup.openai.com`.

[10] R. Sutton and A. Barto *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[11] G. Rivérieulx de Varax *Real-time control of milling machine using reinforcement learning techniques*. Master Thesis, EPFL ICT for Sustainable Manufacturing, 2019.

[12] R. Sutton, D. A. McAllester, S. P. Singh and Y. Mansour *Policy gradient methods for reinforcement learning with function approximation*. Neural Information Processing Systems 12, pages 1057–1063, 1999.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller *Deterministic policy gradient algorithms*. ICML, 2014.

[14] T. P. Lillicrap et al. *Continuous control with deep reinforcement learning*. ICLR, 2016.

[15] S. Fujimoto, H. van Hoof and D. Meger *Addressing Function Approximation Error in Actor-Critic Methods*. ICML, 2018.

[16] T. P. Lillicrap et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. arXiv preprint, arXiv:1801.01290, 2018.

[17] R. Antonova, S. Cruciani, C. Smith and D. Kragic *Reinforcement Learning for Pivoting Task*. arXiv preprint, arXiv:1801.01290, 2017.

[18] J.-M. Engel and R. Babuska *On-line reinforcement learning for nonlinear motion control: Quadratic and non-quadratic reward functions*. IFAC Proceedings Volumes, vol. 47, no. 3, pp. 7043–7048, 2014.

[19] B. Daley and C. Amato *Reconciling -Returns with experience replay.* Advances in Neural Information Processing Systems, 1133-1142, 2019.

[20] X. Peng, M. Andrychowicz, W. Zaremba and P. Abbeel *Sim-to-real transfer of robotic control with dynamics randomization.* IEEE International Conference on Robotics and Automation (ICRA), Brisbane, Australia, May 2018.

[21] T. Zahavy, N. Baram and S. Mannor *Graying the black box: understanding DQNs.* arXiv preprint, arXiv:1602.02658, 2016.

[22] V. Mnih et al. *Human-level control through deep reinforcement learning.* arXiv preprint, arXiv:1602.02658, 2016.

[23] B. A. Francis and W. M. Wonham *The internal model principle of linear control theory.* Automatica (Journal of IFAC), September 1976.

[24] J. Munk *Pretraining actor-critic networks using state representation learning.* Master Thesis, Delft Center for System and Control, April 2016.

[25] L. van der Maaten and G. Hinton *Visualizing data using t-SNE.* Journal of Machine Learning Research 9, 2579-2605, 2008.

[26] B. D. Ziebart, A. Maas, J. A. Bagnell and A. K. Dey *Maximum entropy inverse reinforcement learning.* Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, 2008.