Reinforcement learning-based control system for the SK80 robot

Author
Dominik Hodan

Supervisor
Krištof Pučejdl

CTU
Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Control Engineering
AA4CC

2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hodan Dominik** Personal ID number: **474587**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Reinforcement learning-based control system for the SK8O robot

Master’s thesis title in Czech:

Řídicí systém pro robota SK8O využívající posilované učení

Guidelines:

The SK8O robot is currently using traditional control methods, such as PID and LQR, to drive and balance. This project aims to explore Reinforcement Learning (RL) as an interesting alternative to these approaches, with the goal of replacing and improving the currently used controllers.
To achieve this goal, the following steps will be taken:
1. Develop a RL-based control system for balancing trained on a linear dynamical model of the robot. The resulting algorithm should comply with the computational constraints of the onboard hardware.
2. Test the control system using a 3D model of the robot in a simulation.
3. Deploy the algorithm on the actual robot hardware and validate its performance.
4. Extend the control algorithm to utilize the 3D dynamics of the robot, namely the option to lean sideways.
5. Explore the possible impact of the upcoming European Union Liability AI Directive on the SK8O robot (if it is to be produced commercially).

Bibliography / sources:

[1] SCHULMAN, John, Sergey Levine, Pieter Abbeel, Michael Jordan and Philipp Moritz. „Trust region policy optimization“. International conference on machine learning, 1889–97. PMLR, 2015.
[2] SCHULMAN, John, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. „Proximal policy optimization algorithms". arXiv preprint arXiv:1707.06347, 2017.
[3] LEVINE, Sergey, Aviral Kumar, George Tucker and Justin Fu. „Offline reinforcement learning: Tutorial, review, and perspectives on open problems“. arXiv preprint arXiv:2005.01643, 2020.
[4] PENG, Xue Bin, Marcin Andrychowicz, Wojciech Zaremba and Pieter Abbeel. „Sim-to-Real Transfer of Robotic Control with Dynamics Randomization“. 2018 IEEE International Conference on Robotics and Automation (ICRA). May 2018. pp. 3803–3810. DOI 10.1109/ICRA.2018.8460528, 2018.
[5] HAARNOJA, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel and Sergey Levine. „Soft Actor-Critic Algorithms and Applications“. ArXiv preprint arXiv:1812.05905, 2019.

Name and workplace of master's thesis supervisor:

Ing. Krištof Pučejdl Department of Control Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **19.02.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until:
by the end of summer semester 2023/2024

Ing. Krištof Pučejdl	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration of Authorship

I, Dominik Hodan, declare that this thesis titled "Reinforcement learning-based control system for the SK8O robot" and the work presented in it are my own. I declare that I wrote the presented thesis on my own and that I cited all the used information sources in compliance with the Methodical instructions about the ethical principles for writing an academic thesis.

Signed: _____

Date: _____

Abstract

In this master's thesis, I explore the application of reinforcement learning techniques to balancing and velocity tracking of a bipedal wheeled robot named SK8O. To this end, two distinct training environments are developed: one emulating a linear Segway model and another that performs a full 3D rigid body simulation. These environments are then used to train deep neural networks using the Soft Actor-Critic algorithm and its variants. The resulting controllers are verified in simulations and on the embedded system in the real robot. The work concludes with a consideration of artificial intelligence liability and an introduction to the methods of explainable artificial intelligence.

keywords: reinforcement learning, sim-to-real transfer, mujoco, soft actor-critic, explainable artificial intelligence, SK8O

Abstrakt

V této diplomové práci se věnuji posilovanému učení a jeho využití pro balancování a sledování rychlostní reference kolového dvounohého robota zvaného SK8O. Za tímto účelem jsou vytvořena dvě simulační prostředí: první modeluje linearní aproximaci Segwaye a druhý je plnou 3D simulací systému. Tato prostředí jsou následně použita k tréninku hlubokých neuronových sítí pomocí algoritmu Soft Actor-Critic a jeho variant. Výsledné regulátory jsou ověřeny v simulacích a na vestavném systému ve skutečném robotovi. Práci zakončuje zvážení odpovědnosti umělé inteligence spolu s úvodem do metod oboru vysvětlitelné umělé inteligence.

klíčová slova: posilované učení, transfer ze simulace na skutečný systém, mujoco, soft actor-critic, vysvětlitelná umělá inteligence, SK8O

Contents

Acronyms

AA4CC	Advanced Algorithms for Control and Communications
AI	Artificial Intelligence
COM	center of mass
CTU	Czech Technical University in Prague
CV	Computer Vision
D2RL	Deep Dense Architectures in Reinforcement Learning
DL	Deep Learning
DOF	Degree of Freedom
EU	European Union
FEE	Faculty of Electrical Engineering
HER	Hindsight Experience Replay
i.i.d.	independent identically distributed
IMU	Inertial Measurement Unit
LQR	Linear Quadratic Regulator
LSTM	Long Short-Term Memory
MDP	Markov decision process
ML	Machine Learning
MLP	Multilayer Perceptron
NLP	Natural Language Processing
NN	Neural Network
POMDP	Partially Observable Markov Decision Process
RCI	Research Center for Informatics, CTU Prague
RL	Reinforcement Learning
RNN	Reccurent Neural Network
SAC	Soft Actor-Critic
SGD	Stochastic Gradient Descent
SLAM	Simultaneous Localization and Mapping
SOTA	state of the art
W&B	Weights and Biases
XAI	Explainable Artificial Intelligence
XRL	Explainable Reinforcement Learning

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my parents, without whose unconditional love and support I would not be able to dedicate this much time and energy into my studies. My thanks extend to the rest of the family for their patience and understanding when they did not hear from me for the past few months.

I am deeply grateful to my girlfriend, whose presence and support provided me with solace and much-needed respite. Especially in the weeks leading up to the deadline, moments with her were a cherished escape from the demands of this thesis for my mind.

Of course, I cannot overlook the contributions of my supervisor, Krištof Pučejdl and my unofficial co-supervisor, Martin Gurtner, who both went above and beyond in their duties, sharing their contagious excitement for the project when I needed it most. It may sound cliché but I truly do not think I would have been able to accomplish such amount of work without their combined support and mentorship. The following text would also be noticeably harder to read without Krištof's keen eye.

Other members of the AA4CC group also deserve a mention. I would especially like to express my appreciation of Šimon Lehký and Dominik "Fomča" Fischer for their positive energy in our shared experience. Special thanks go to Loi Do for his valuable tips on Ipe and the visual aspects of this document in general.

I would furthermore like to acknowledge the assistance of prof. Robert Babuška who provided me with insightful tips on practical reinforcement learning, and Dr. Jakub Mareček, whose pointers regarding the field of AI explainability saved me a considerable amount of time that could have been reinvested in other aspects of the thesis.

I express my sincere appreciation to the Research Center for Informatics, CTU, for kindly providing access to the supercomputer where my programs have lived for years worth of computing time while performing the experiments presented in this thesis.

To end on a lighter note, I would also like to credit the author of the cover image: the Stable Diffusion model, along with the cubist painters whose work served as its inspiration.

Chapter 1

Introduction

In late March 2023, news outlets around the world reported about the Future of Life Institute, a nonprofit organization, issuing an open letter with a rather plain title – *Pause Giant AI Experiments*. This text, signed by the likes of Steve Wozniak and Elon Musk, warns that models more capable than GPT-4 "should be developed only once we are confident that their effects will be positive and their risks will be manageable" [1]. Even if we step back from this affair, it is clear that Artificial Intelligence (AI) is becoming more influencial to the society at large by the year.

One field that seems to resist the advance of AI is control engineering. There are two main obstacles hindering its adoption. First, while the recent success of AI has been largely achieved by the ever-larger deep neural models, those used in Reinforcement Learning (RL), which aims to solve similar problems as control, are typically much simpler and less capable due to poor sample efficiency.

The second reason is the black-box nature of deep models and its implications for their safety and guarantees. Indeed, while a chatbot claiming that $2 + 2 = 5$ might conjure up a smile and perhaps a memory of a certain old book, an airplane making the same mistake may have more serious ramifications.

This thesis is, in a sense, a case study on RL applied in real-life. We will explore whether the current methods can beat a classical controller in a standard task. We will also see whether the state of the art (SOTA) can provide us with any assurances about its behavior.

1.1 Outline

In Chapter 2, you can read about SK8O, the robotic test subject of this work. Apart from a standard description of its features, we will focus on its computational capabilities, which may be critical for running Machine Learning (ML) models. I also cover the related work that has been done on the robot – mainly on its modeling.

In Chapter 3, we will discuss Reinforcement Learning. After a short introduction, I cover the main method in this thesis, Soft Actor-Critic (SAC), and a large variety of possible extensions. We will also mention many possible issues with RL and how SOTA methods attempt to tackle them.

Chapters 4 and 5 each deal with a different way of simulating the robot's dynamics: as a linearized Segway model and as a full 3D rigid body simulation. We will then see how the algorithms introduced in the preceding chapter learn to stabilize the respective systems and to make them follow a velocity reference. At the end of each chapter, the best performing controllers are evaluated. The results are then validated on the real robot in Chapter 6.

We finish by discussing a possible upcoming European Union (EU) directive on AI liability in Chapter 7. There, I also review several techniques of Explainable Artificial Intelligence (XAI) that could potentially be used to comply with the directive in future applied research and in the industry.

Chapter 2

SK8O Robot

The centerpoint of this thesis is SK8O, a bipedal wheeled robot, meaning that it has two legs, at the end of which are wheels. It was designed and constructed in the Advanced Algorithms for Control and Communications (AA4CC) group at Faculty of Electrical Engineering (FEE) Czech Technical University in Prague (CTU) and its topology (particularly the way its legs include a kinematic loop) was inspired by a similar robot developed at ETH Zürich known as Ascento.[1] To a large extent, the robot is 3D-printed with a focus on keeping the costs down and using off-the-shelf components, as it is planned to be open-sourced in the near future.

A photo of the robot is shown in Figure 2.1. If you would like to see the robot in action, an interview with its creators, Krištof Pučejdl and Martin Gurtner, is available on YouTube (in Czech).[2]

2.1 Related work

The robot has already shown its merit as a learning tool – apart from this thesis, two have been already defended and there is concurrent work being done as well. Hopefully, its popularity among students will continue in the future and particularly the simulator developed in Chapter 5 can be reused.

An important source of information for this work was the thesis of Adam Kollarčík. [2] It contains a more detailed technical description of the robot than the one provided here, as well as system identification and the derivation of the segway model covered in Chapter 4. This work also developed a Linear Quadratic Regulator (LQR), which will be used as the baseline controller in this thesis.

Higher level control followed soon after. In [3], SK8O was equipped with an Intel Realsense camera, which was used to perform Simultaneous Localization and Mapping (SLAM). The reference is only included here for completeness and will not be used in any other way.

[1] https://www.ascento.ch/
[2] https://www.youtube.com/watch?v=-7dsug0FtP8

Figure 2.1: Photo of the SK8O robot

2.2 Description

Now, we will take a look at some of the features of the robot relevant for this thesis, be it for modeling or control. Should the reader be interested in a more detailed description, they may consult [2].

Actuators

The configuration of each leg can be controlled by a motor at the *hip*. Due to a closed kinematic chain, each leg only has one degree of freedom (ignoring the rotation of the wheel). The dimensions of the links were designed so that changing the angle at the hip translates to the wheels moving approximately along a vertical straight line.

Often in this text, we will allow for another anthropomorphization by referring to the revolute joint between the hip and the wheel as the *knee*. By bending the knees, SK8O is able to move its center of mass up and down and potentially pass under obstacles. Furthermore, a rapid extension of both legs allows the robot to jump. The knees contain a torsion spring that partially counteracts gravity.

Additionally, each of the wheels is powered by its own separate motor, bringing the number of system inputs to four.

All four of the brushless DC motors are the same, position or torque-controlled at a frequency of 40 kHz – at least an order of magnitude faster than any controller that might interact with them. Therefore, we will assume ideal behavior when modeling them. The torques are software-limited to 1 N m. However, the hip actuators are geared up by a factor of 16.5 to provide sufficient torque.

Sensors

An Inertial Measurement Unit (IMU) provides linear accelerations and angular velocities at 1 kHz. Its outputs are then fused to obtain Euler angles of the robot's main body. Additionally, each of the motors provides its position, velocity and the torque being applied.

2.2.1 | Computational resources

The current boom of Deep Learning (DL) can be partially attributed to rising computational power – Neural Networks (NNs) are notoriously demanding. Even though RL typically employs much smaller and simpler models than other fields of ML where they have been applied, such as Natural Language Processing (NLP) or Computer Vision (CV), this is something that we will need to keep in mind, should we want to deploy the models on the real robot. Therefore, we will take a closer look at SK8O's "brains".

The plural at the end of last paragraph was no mistake – the robot is controlled by a pair of computers. As of right now, any real-time operation is performed on a *Teensy 4.0*, which runs the feedback controller introduced in Section 4.1.1. More demanding tasks that do not require real-time computation can be offloaded to an *Odroid N2+* board. This board also facilitates remote interaction with the robot over Wi-Fi and/or an Xbox controller. A schematic diagram can be seen in Figure 2.2.

There was also work being done by Petr Brož to enable real-time control on the *ODroid* board, which was concurrent with this thesis. Thanks to his advances, it is possible to deploy the models to the ODroid board, which is significantly less resource contrained. Additionally, it runs Ubuntu Linux and makes it easy to deploy models for example in the ONNX format.

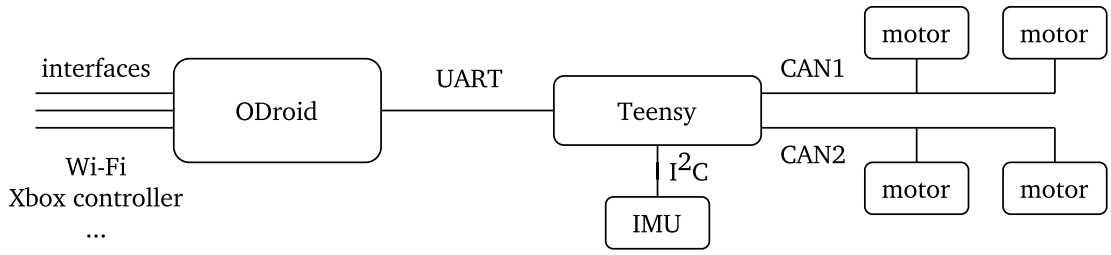


Figure 2.2: Diagram of the electronics, reproduced from [2]

USB accelerators

Should we need to increase computational power of the ODroid, a possible solution would be to use a USB inference dongle designed to efficiently evaluate neural networks when they are deployed. Two popular choices exist – *Coral USB accelerator*[3] (supports the TFLite format) and *Intel Neural Compute Stick 2*[4] (supports TFLite, Pytorch, ONNX and others). Unfortunately, as of early 2023, the Coral accelerator is facing shortages and is not in stock anywhere and Intel has discontinued production of the latter altogether.

[3] https://coral.ai/products/accelerator
[4] https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html

2.3 Modeling

Even though the methods explored later in this thesis can in principle be applied even with data collected on the real robot (they are so-called *off-policy* methods) as demonstrated for example by [4], it is still beneficial to have a simulator available for testing. This also has the benefit that when SK8O is open-sourced, having a functioning simulation can be a very convenient addition to the whole package.

In a previous work done by Adam Kollarčík, several mathematical models of the robot were devised [2]. I implemented two of the models presented there as a part of this thesis. In particular, these were the Segway model and a full numerical simulation of the 3D model. Unfortunately, both models had to be completely reimplemented in different languages/systems.

There were two main reasons why we decided to create two models. First, the linearized and discretized version of the Segway model is very simple to implement in Python (the original was implemented in MATLAB), so it was fast to get the training up and running. Second, because of its simplicity, it is also very light-weight, making it good for trying out different algorithms. On the other hand, there are numerous benefits of an accurate full simulation, such as the chance to verify controllers before they are deployed onto the real robot, where they may cause damages.

In Chapter 4, we will take a look at the linear Segway model and in Chapter 5, we will discuss the full 3D numerical simulation implemented in MuJoCo.

Chapter 3

Reinforcement Learning

In this chapter, we will set SK8O aside for a moment and focus on the main objective of this thesis. As the title suggests, we will apply methods of Reinforcement Learning to control the robot we introduced in the previous part.

In RL, the goal is to deduce what actions to take in an environment to maximize numerical rewards given to us at each time step [5]. The online problem formulation, in which the agent can interact with the environment to gauge the effects of its actions, should be very familiar to us – a child learns to conform to societal expectations (such as not taking the question "How are you?" at face value in some cultures) in such a way that receives the best response from the people around them, subconsciously solving a very similar task.

We begin in Section 3.1 by quickly defining the terminology that will be used later on. In Section 3.2, we will go over one of the most popular model-free algorithms, SAC, as well as its recent modifications. In Section 3.3, we will explore methods that attempt to speed up the training and finish by a slight detour, in which we will consider the (dis)advantages of model-based methods in Section 3.4.

3.1 Formal Setting

Let us quickly go over the problem RL aims to solve – the (Partially observable) Markov decision process (POMDP). This will also naturally let us properly define the quantities of interest in a POMDP and the chosen notation.

The environment

The cornerstone of each RL problem is the environment, something a control engineer might loosely imagine as a dynamical system. It is typically either inherently discrete in time or a sampled process and we model it as a Markov decision process (MDP). In the simplest relevant case, it can be specified by a tuple $(\mathcal{S}, \mathcal{A}, \tau, \rho)$. To illustrate their meaning, let us go over what happens in a single time instance.

At the beginning of time t, the environment finds itself in *state* s_t – an element of its state space $\mathcal{S}$, where $\dim \mathcal{S} = S$. As this thesis is oriented towards control engineering,

we will only consider continuous observation spaces, i.e., $\mathcal{S} \subseteq \mathbb{R}^S$. Upon receiving an input (an *action*) $a_t \in \mathcal{A} \subseteq \mathbb{R}^A$, it transitions to a new state s_{t+1} according to its transition function

$$\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{S}. \tag{3.1}$$

If we allow the environment to be stochastic, which is often desirable when modeling a real system, we say

$$\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S}) \tag{3.2}$$

and understand the new state to be sampled from the probability distribution, $s_{t+1} \sim \tau(s_t, a_t)$. Lastly, the environment produces a reward according to its reward function

$$\rho : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R} \tag{3.3}$$

as $r_t = \rho(s_t, a_t, s_{t+1})$. When it will not lead to confusion, we will use the standard shorthand notation $(s, a, r, s') := (s_t, a_t, r_t, s_{t+1})$.

Only a slight complication (from a notational point of view) is to consider a Partially Observable Markov Decision Process. In that case, the environment is specified by $(\mathcal{S}, \mathcal{O}, \mathcal{A}, \tau, \rho, \omega)$. The extra function

$$\omega : \mathcal{S} \to \mathcal{O} \subseteq \mathbb{R}^o \tag{3.4}$$

takes the environment state and returns an $o-$dimensional *observation*. Of course, we can again consider a stochastic version of the observation function (e.g. to model measurement noise), which would be analogous to the stochastic transition from Equation (3.2).

Note that all variables at time $t+1$ depend only on variables at time t – this "lack of memory" is a crucial feature of MDPs. This still holds true for POMDPs. However, because the true state is hidden from us, we might need to infer it based on a sequence of observations, which need not be that short.

The agent

In online RL, an agent interacts with the environment in discrete timesteps. It can be represented by a policy π parametrized by ϕ, denoted jointly π_ϕ, though we will frequently omit the subscript for brevity.

Often, a policy is simply a function $\pi : \mathcal{O} \to \mathcal{A}$. However, the algorithms we will use throughout this text produce stochastic actors. Therefore, our policies will in fact generate a probability distribution over the action space, which can be denoted by

$$\pi : \mathcal{O} \to \mathcal{P}(\mathcal{A}). \tag{3.5}$$

In a slight abuse of notation, we will sometimes write $\pi(a_t \mid s_t)$ as "the likelihood of sampling action a_t from distribution $\pi(s_t)$".

The above summary of the interaction between the agent and the environment is depicted in Figure 3.1.

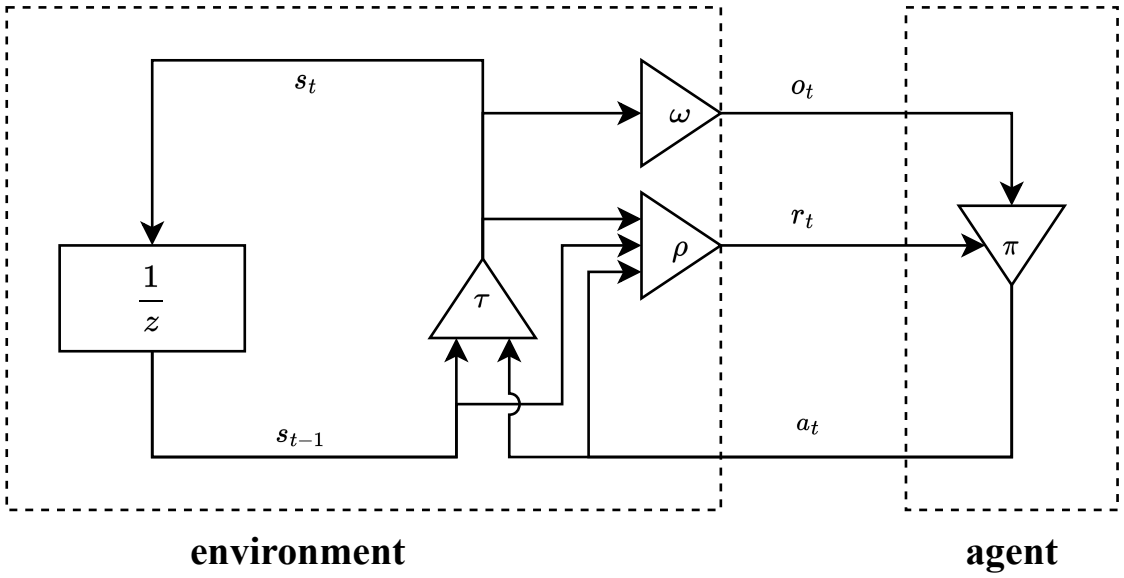


Figure 3.1: An illustration of the interaction between the agent and the environment in a single time step

3.1.1 | Q-learning

As we have said before, in RL, the goal of the agent it to maximize a function of the rewards. Commonly, the quantity of interest is the *return*, defined as

$$R_T = \sum_{t=T}^{\infty} \gamma^t r_t, \tag{3.6}$$

where γ is known as the *discount-rate*, typically chosen as $\gamma \in [0.9, 1]$. This factor affects the effective horizon length.

Furthermore, there are two common functions of interest that tie the return R to the states, actions and policy we defined previously. Namely, they are the *(state-)value function*, which describes the expected return when starting in a particular state and behaving according to policy π,

$$V^\pi : \mathcal{S} \to \mathbb{R}, \quad V^\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)}[R_0 \mid s_0 = s], \tag{3.7}$$

and the $Q-$*function*, which additionally considers the "desirability" of a specific initial action,

$$Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}, \quad Q^\pi(s, a) = \mathbb{E}_{a_t \sim \pi(s_t), t>0}[R_0 \mid s_0 = s, a_0 = a]. \tag{3.8}$$

If optimal actions (those that maximize the expected return) are taken, we get the optimal V and Q functions,

$$V^*(s) = \max_\pi \mathbb{E}[V^\pi(s)], \tag{3.9}$$

$$Q^*(s, a) = \max_\pi \mathbb{E}[Q^\pi(s, a)]. \tag{3.10}$$

Notice that knowing Q^* theoretically enables us to always choose the optimal policy as

$$\pi^*(s) = \arg\max_a Q^*(s,a), \tag{3.11}$$

though this is complicated by the fact that finding the maximum is often not feasible.

There are two important observations about the relationship of the two functions:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)], \tag{3.12}$$
$$V^*(s) = \max_a Q^*(s,a). \tag{3.13}$$

Crucially, the optimal functions satisfy the Bellman equations:

$$V^*(s) = \max_a \mathbb{E}_{s\sim\tau}[\rho(s,a,s') + \gamma V^*(s')], \tag{3.14}$$
$$Q^*(s,a) = \mathbb{E}_{s\sim\tau}[\rho(s,a,s') + \gamma \max_{a'} Q^*(s',a')]. \tag{3.15}$$

Often, such as in the SAC algorithm, we perform iteration over a similar set of equations,

$$V^\pi(s) = \mathbb{E}_{a\sim\pi,s'\sim\tau}[\rho(s,a,s') + \gamma V^\pi(s')], \tag{3.16}$$
$$Q^\pi(s,a) = \mathbb{E}_{s'\sim\tau}[\rho(s,a,s') + \gamma \mathbb{E}_{s'\sim\pi}[Q^\pi(s',a')]], \tag{3.17}$$

to approximate the optimal functions.

The family of methods that set out to find (or approximate) the $Q-$function bears an impressively descriptive name: $Q-$learning. Notice that for the $Q-$function iteration in Equation (3.17), the action a need not be generated by the policy π. This ability to "learn from the mistakes of others" is a major advantage of these so-called off-policy methods. In practice, we typically create a *replay buffer*, where past transitions (s,a,r,s') are stored for later use.

3.2 | Soft Actor-Critic

In the 2010's and early 2020's, many new algorithms for continuous RL have been developed. One of the most popular algorithms currently is called Soft Actor-Critic, introduced in [6] and later improved upon by the same authors in [7].

SAC outperforms other vanilla algorithms in most benchmarks. For example, the benchmarks of the RL library Tianshou show that SAC achieves the best score in all but one of the standard OpenAI Gym MuJoCo tasks – and in the single case where it does not (the Hopper environment), it achieved 99.4% of the best result [8].

Similarly, [9] found SAC to be the "best performing across the board" of the algorithms tested. There, the dominance was not as significant and another off-policy model-free algorithm called TD3 outperformed it at a few problems. However, it was often at the cost of higher variance between different runs. For the limited amount of time I had, I decided to use SAC and not compare it with other vanilla algorithms here (though I did try two other algorithms, A2C and TD3, which performed worse in preliminary tests).

3.2.1 | Algorithm overview

Because SAC and its variations will be applied extensively throughout the rest of this thesis, let us briefly go over the key parts of the algorithm. All of the information in this section is taken from the original paper [7], and the reader is encouraged to consult the source for more details.

The algorithm slightly modifies the optimization problem from Equation (3.6) by adding a constraint, attempting to find the optimal policy π^* defined as

$$\pi^* = \arg\max_{\pi \in \Pi} \mathbb{E}_{a\sim\pi}\left[\sum_{t=0}^{T} r(s_t, a_t)\right] \tag{3.18}$$

$$\text{s.t.} \quad \mathbb{E}\big[-\log\left(\pi_t(a_t \mid s_t)\right)\big] \geq \mathcal{H}, \quad \forall t, \tag{3.19}$$

where $\mathcal{H}$ is the desired minimum expected entropy, typically set to $\mathcal{H} = -\dim \mathcal{A}$. Intuitively, the reason for this choice is to scale the "amount of randomness" with the number of actions, as discussed in [10].

Even though SAC can be considered a $Q-$learning algorithm, there are a few changes. It has been observed that using only a single $Q-$function approximator leads to an overestimation of action-value pairs (predicting higher future returns). Therefore, SAC employs a pair of $Q-$networks for a more accurate estimate, which we will denote Q_{θ_1} and Q_{θ_2} (or Q_1 and Q_2).

Another "trick" is that for the iteration in Equation (3.17) each parameter is updated using yet another $Q-$function that is the running average of itself, called the *target* function, denoted by $Q_{\bar{\theta}_i}$. This was again shown to stabilize learning [11].

The two $Q-$networks are updated via Stochastic Gradient Descent (SGD) by minimizing the loss function

$$L_{Q_i} = \mathbb{E}\left[\left(Q_{\theta_i}(s_t, a_t) - \left(r(s_t, a_t) + \gamma V^{\pi}_{\bar{\theta}_1, \bar{\theta}_2}(s_{t+1})\right)\right)^2\right], \tag{3.20}$$

where the value function is computed as

$$V^{\pi}_{\bar{\theta}_1, \bar{\theta}_2}(s_{t+1}) = \mathbb{E}_{a_{t+1}\sim\pi}\left[\min_{i=1,2} Q_{\bar{\theta}_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\phi(a_{t+1} \mid s_{t+1})\right], \tag{3.21}$$

where α is a *hyperparameter* of the algorithm known as the *temperature*.

Their targets are updated simply by

$$\bar{\theta}_i \leftarrow (1-\tau)\bar{\theta}_i + \tau\theta_i, \tag{3.22}$$

where $\tau \in (0, 1)$, typically chosen as $\tau = 0.05$.

The policy is updated by minimizing

$$L_\pi = \mathbb{E}\left[\alpha \log \pi_\phi(a_t \mid s_t) - \min_{i\in\{1,2\}} Q_{\theta_i}(s_t, a_t)\right]. \tag{3.23}$$

In [7], the algorithm was extended to automatically adjust the temperature α by further minimizing

$$L_\alpha = \mathbb{E}_{a_t \sim \pi}[-\alpha \log \pi(a_t \mid s_t) - \alpha \mathcal{H}]. \tag{3.24}$$

The full algorithm is outlined in Algorithm 1. Notice that in practice, we approximate the expectations in the above equations by sampling minibatches from the replay buffer denoted by $\mathcal{D}$.

Algorithm 1: Soft Actor-Critic

initialize $\alpha, \theta_1, \theta_2, \phi$ to random values
initialize replay buffer $\mathcal{D}$
copy parameters $\theta_i' \leftarrow \theta_i$,
repeat
  reset environment and sample new state s
  for $n = 1, \ldots, N$ **do**
    select action $a \sim \pi(s)$
    perform action a, receive s', r, `truncated, terminated`
    store new transition $\mathcal{D} \leftarrow \mathcal{D} \cup (s, a, s', r)$
    sample a random batch B of transitions from buffer, $B \sim \mathcal{D}$
    $\theta_i \leftarrow \theta - \lambda_\theta \nabla_{\theta_i} L_{Q_i}$
    $\phi \leftarrow \phi - \lambda_\phi \nabla_\phi L_\pi$
    $\alpha \leftarrow \alpha - \lambda_\alpha \nabla_\alpha L_\alpha$
    update target networks using 3.22
    if *terminated or truncated* **then**
      break
    end
  end
until *satisfied*

As for the structure, the vanilla algorithm uses simple Multilayer Perceptrons (MLPs) with linear layers alternating with ReLU[1] activation functions, as shown in Figure 3.2. Below, we will also explore different strucures.

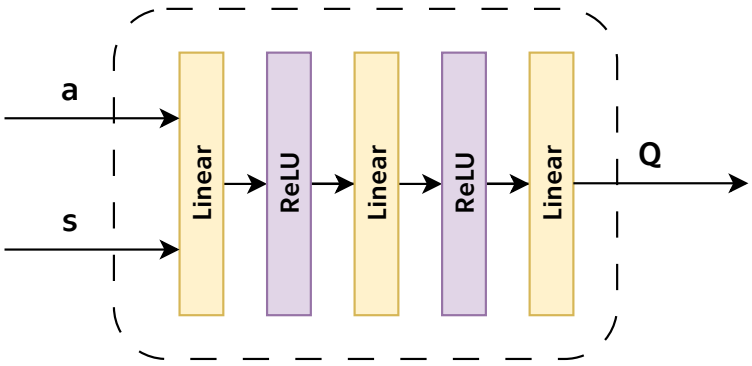


Figure 3.2: SAC uses simple MLPs for both the $Q-$networks and the policy π.

[1] https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

3.2.2 | Improving sample efficiency

Much work has been done in the past few years to improve this algorithm further. Despite using two critic networks, it has been shown that it still suffers from $Q-$value overestimation. Therefore, many methods have been devised that attempt to alleviate this issue.

Recently, REDQ ([12]) has shown that using an ensemble of critics and then using a random subset of them for Equation (3.23) significantly improves the sample efficiency of the algorithm, at the cost of much greater computational demands during training. Later, an algorithm called DroQ showed similar performance with a much smaller footprint, sufficing with the same amount of networks as the original algorithm. [13]

DroQ uses two regularization techniques popular in Computer Vision. Namely dropout layers, which randomly drop a certain percentage of neuron connections during training to prevent overfitting [14] and layer normalization, which rescales the feature vectors that propagate through the network. This allows the $Q-$networks to be updated 20 times per environment interaction. The full structure of the $Q-$function is depicted in Figure 3.3.

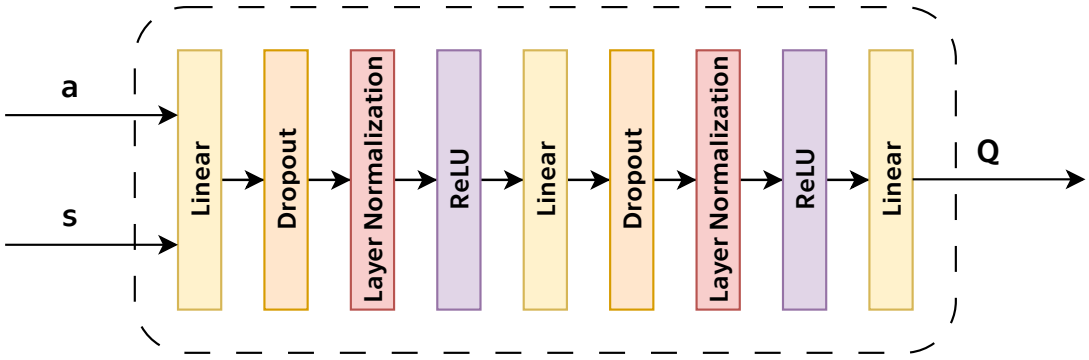


Figure 3.3: Structure of the Q-function in DroQ

Even though sample efficiency is not strictly necessary for our application (we can train in simulation), it can still be interesting to see whether this approach can learn faster, so I decided to implement and test DroQ. Additionally, should we need fine-tuning after deployment to SK8O, sample efficiency can be beneficial. It has even been shown, that this algorithm is able to learn to walk on a real robot "in the wild" within 20 minutes [4], which is a very encouraging result for RL in my opinion.

In the future, it may be interesting to test another improvement of REDQ – the AQC algorithm from [15], showing both better sample efficiency and asymptotic performance. This algorithm uses multiple heads (network outputs that share the same low-level features) for each of the $Q-$networks in the ensemble, reducing the demands as well, though not as significantly as DroQ.

3.2.3 | Network structure

In model-free RL, improvements are typically made by introducing more sophisticated algorithms. Indeed, the network used most commonly with SAC is a MLP, which was published already in 1986 [16]. This is in stark contrast to other fields of ML, where complicated network architectures are being developed – compare the block diagram of a Transformer (introduced in [17]), an architecture popular in NLP, with the MLP.

As someone who has worked in CV, I was curious to see whether I can replicate the results from [18]. Their method named D2RL uses skip-connections as seen in 3.4, which, according to their experiments, drastically improve the performance of deeper models in RL and such deeper models beat vanilla SAC in 4/5 MuJoCo environments. The best performing models had 4 layers, which is double the amount in SAC, yet the network is still small enough to be deployed on many embedded devices, such as SK8O's ODroid.

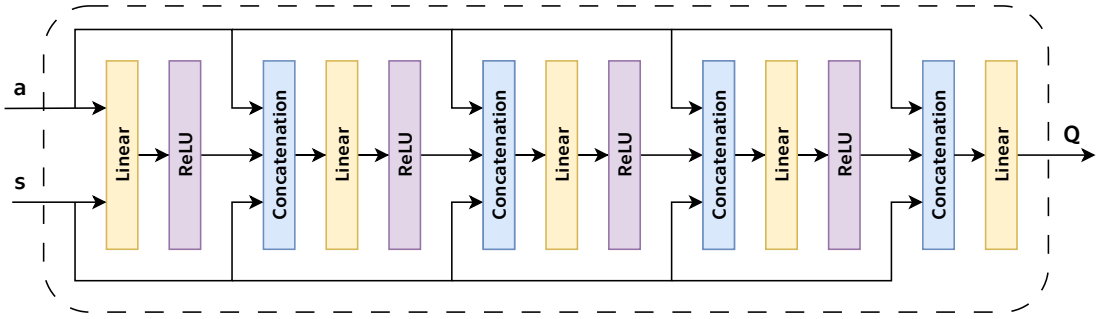


Figure 3.4: Structure of the $Q-$network in D2RL. The agent has an analogous structure.

3.2.4 | Domain randomization

One major obstacle when training in a simulated environment is the leap to the real system. The inaccuracies in our mathematical model can cause poor performance of the trained agent if it is not able to generalize well. Therefore, many methods have been developed to curb this problem. [19] lists a plethora of such approaches. However, we will only focus on one – *domain randomization*.

The idea is simple: if we need the agent to be robust against model inaccuracies, we train it on many similar environments with slightly modified simulation parameters. This method is rather easy to implement as we have our own simulation. As a bonus one can even intrinsically include parameter uncertainties if they are known.

The authors of [19] test this approach on a robot arm that is tasked to push an object on a flat surface. Although they achieve the highest success rate on the real robot using a recurrent Long Short-Term Memory (LSTM) network, even the MLP shows a drastic improvement over the agent that did not learn with domain randomization. They show that the performance can be further improved if instead of only showing the agent the last observation, one concatenates the last n observations. This is known as *frame-stacking*.

Domain randomization was also tested in [20], which is a system that is ever-so-slightly similar to SK8O – a robot with four wheeled legs.

3.3 Exploration

When the observation and action spaces are large, it may be hard to "stumble upon" even a slightly reasonable policy, which can then be refined into a near-optimal one. This is true especially for environments with sparse rewards, i.e., when only the terminal states have non-zero rewards – the agent may never find such a state during training!

Some of the advances in RL have been due to better exploration strategies that help mitigate this problem. In the DDPG algorithm from 2016, temporally correlated noise is used (in particular, the Ornstein-Uhlenbeck process) [21]. In contrast, the authors of SAC use a stochastic actor and include entropy in the loss function to promote exploration and argue that this is an important ingredient that leads to lower brittleness (sensitivity to hyperparameters) of the algorithm [6].

An orthogonal approach to solve this problem is to modify the experience replay buffer to include samples from successful policies. We will now take a look at two such methods.

3.3.1 Hindsight experience replay

One popular approach, introduced in [22], is called Hindsight Experience Replay (HER). The idea is rather simple – if the agent did not succeed in the required task, find a similar task where it did. One of the problems the authors use to illustrate the method is puck sliding – the goal is to use a robotic arm to move a puck to a specified position of the hand's reach, as illustrated in Figure 3.5.

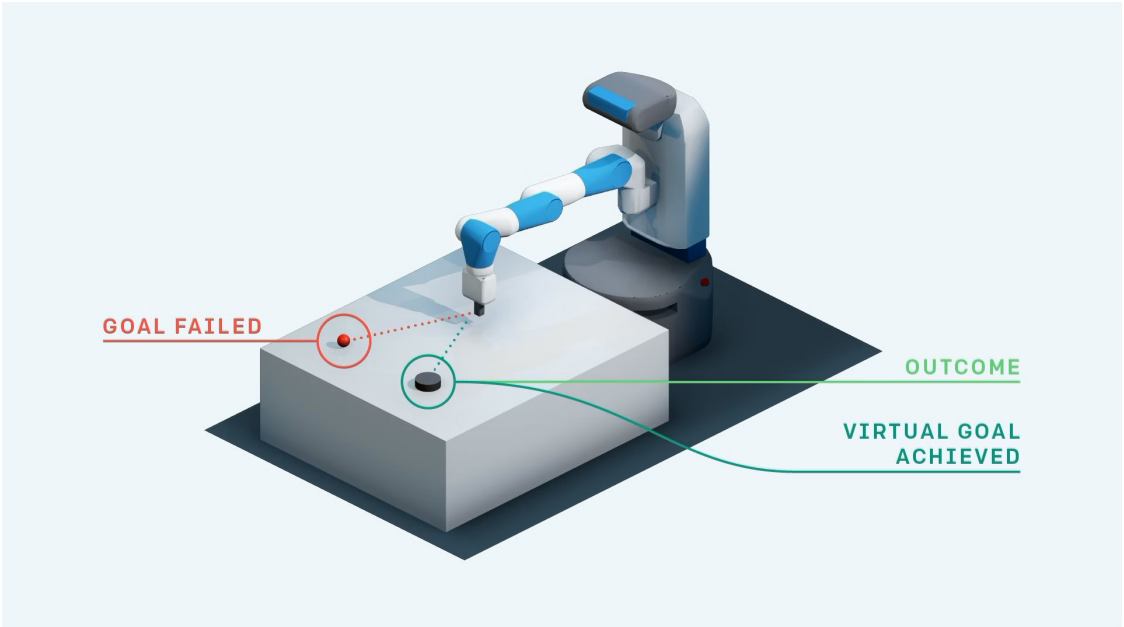


Figure 3.5: Puck sliding example of HER, from [23]

Even though HER has shown promising results, I decided not to use it. The main reason is that unlike in the puck sliding problem (and other tasks I have seen where HER has shown significant improvements), not all final states are created equal in our case of

an unstable system. In fact, only a minuscule subset of states is favorable due to the need to keep the acceleration near zero (more on that later in Section 4.3.1 and 5.3.3).

Yet another reason against using HER in our problem is the fact that it does not play well with dense rewards, as shown already in the original paper. While from a different point of view, this can be seen as one of its benefits (reward shaping requires domain-expertise), we already have an LQR controller for this system, so we have an idea about what features are important and what to penalize. In fact, a large portion of the reward we will use is potential, as recommended by [24].

3.3.2 | Hierarchical learning

When solving problems in the "classical" way (for lack of a better word), it is often beneficial to decompose the problem into smaller, more manageable tasks. Even though it is tempting to try to use the expressivity of NNs to get a plug-and-play all-encompassing solution, this approach has been tested in ML as well.

An interesting strategy is shown in [25], where conventional control is used to ensure stability of the system, and RL was used to compensate for disturbances and uncertainties based on measured data. Such combined control law outperformed both methods used separately.

For a purely DL approach, in [26], the authors learn a set of nested policies in parallel. To illustrate their approach, they showcase a problem where the Ant has to move into a different room in the world map, as shown in Figure 3.6. The outermost policy obtains the true goal position as well as the standard observation and outputs a subgoal – a target position that is easier to achieve. This subgoal, along with the observation, is then used as the input for the next policy. The final policy has a much closer subgoal and outputs the action. To solve the issues with sparse rewards for the lower level policies, HER is used (introduced above in Section 3.3.1).

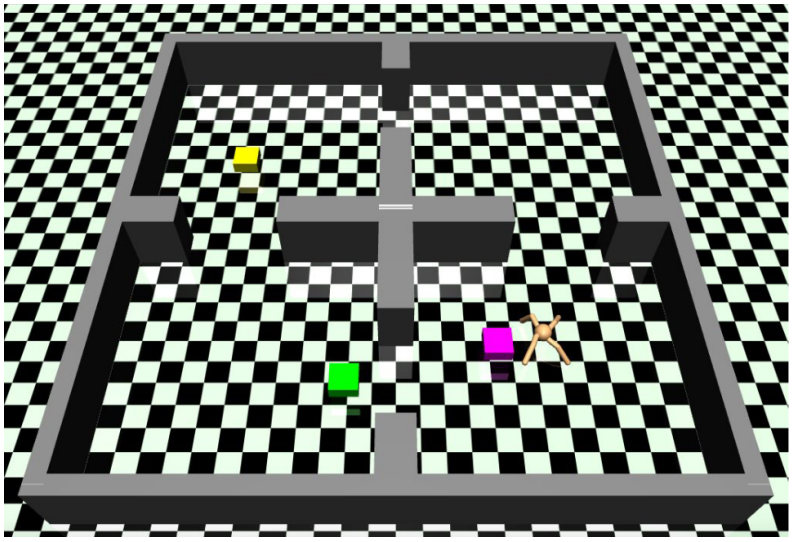


Figure 3.6: Illustration of the HAC algorithm from [26]. The goal position is in yellow and the subgoals generated by the first two policies are in green and pink. Image from the original paper.

Although the tasks presented in the paper solve trajectory planning, it might be an interesting exercise to reformulate this for our velocity tracking problem, where each policy would instead output a velocity reference subgoal instead of a position. Likewise, the hip control could be decomposed into two separate problems: chosing hip position and actually achieving it. I leave this as a possible future research direction.

3.3.3 | Imitation learning

A whole family of algorithms known as imitation learning use an *expert* (controller) to guide the agent during training. The most straightforward way to do this when we work with off-policy algorithms is to generate sample trajectories using the expert, save these into the replay buffer and let the algorithm learn on the dataset, effectively converting the RL problem into supervised learning. This is known as Behavioural cloning and often shows poor performance in practice. Typically, this is attributed to two issues: first, the samples from a trajectory are certainly not independent, and hence the independent identically distributed (i.i.d.) assumption of ML is broken. Second, due to imperfect learning, the agent might encounter situations the expert never did, and it might not be able to generalize properly [27].

Numerous algorithms try to improve this approach e.g., by

- querying the expert online on trajectories generated by the policy (DAgger, [27]),
- slowly shifting the generated actions from the expert policy to the learned policy (SMILe, [28]),
- employing the generative adversarial network training structure to generate actions similar to the expert (GAIL, [29]).

In spite of many such methods being implemented in the `imitation` Python package, I think that they are not what we are looking for – even though realistically, we cannot expect the RL to outperform the current controller used on SK8O on all fronts, we would still like it to have more tricks up its sleeve, in particular leaning into curves.

I personally liked the approach used in [30], where they use a simple PID controller to help train an RL controller – at each step, either the action of the policy or the classical controller is chosen. In their paper, the critic chooses the action to be taken based on the respective $Q-$values. However, I used a simpler approach where I have an exponentially decaying probability of choosing the action proposed by the controller. Additionally, I implemented Behavioural cloning, except the transitions experienced by the agent during training are also added to the replay buffer to allow the agent to surprass the baseline controller.

3.4 | Model-based algorithms

So far, we have only considered model-free algorithms. That is, those that do not (attempt to) explicitly use a model of the environment other than by interacting with it. However, this is not to say that model-based algorithms are not an active branch of research – the reader might be familiar, for example, with Google's MuZero [31] or DreamerV2, in which the agent is trained on a learned latent-space representation of the environment [32]. There have also even been attempts to bring Transformers, an architecture popular in NLP, to model-based RL [33].

Despite their successes in the Atari benchmark suite, model-based methods have yet to show as much success in the AI Gym MuJoCo benchmark, which is more relevant for us. Often, they assume a discrete aobservation and/or action space, so they are not directly transferable to control tasks. Perhaps more importantly, however, they typically utilize much larger models. One exception to both of these, which could be tested in the future, is a model-based algorithm known as MBPO which uses SAC as its policy optimization algorithm [34].

3.4.1 | Variational Recurrent Models

As we will see in the following chapters, including several observations has a notable benefit when uncertainty is present in the system. However, deciding on how many past observations to include is yet another hyperparameter to tune, and I personally do not find that to be an elegant solution. After all, the Kalman filter does not work with a fixed set of observations to estimate the true state either, so perhaps it might be worthwhile to look at approaches that use Reccurent Neural Networks (RNNs).

An excellent opinion article I read about the (lack of) success of RL mentioned that one of the problems of RL is the need for rather complex tooling compared to other fields of ML [35]. This is even more true for recurrent techniques for off-policy learning. This is due to a process called *burn-in*, in which the internal states of the RNN process several previous observations before the one we need to make a prediction about. This requires more complex sampling from the replay buffer and slows down training by itself in addition to using more complex networks, which also take longer to train.

Despite these drawbacks, I was intrigued by the architecture proposed in [36], which uses a recurrent variational autoencoder. A control engineer might think of it as a state observer, except at no place do we specify the state – the network is left on its own to find a good latent space representation that can be used to encode the observation and to predict the next one. This latent representation is then used, along with the observation generated by the environment, as inputs to the networks in SAC, as shown in Figure 3.7.

I was hoping that such a model could learn to perform model estimation "on-the-fly" and thus be more robust against uncertainty in the model. I reimplemented their model and replicated their results on partially observable MuJoCo tasks, such as the Pendulum environment with only velocities available.

Unfortunately, when I tested the algorithm even on a simple integrator environment, where $s_{t+1} = s_t + a_t$ and $\mathcal{A} = [-1, 1]$, the predictor was unable to predict the next state with a consistent error below 0.5. The authors themselves note that the "prediction

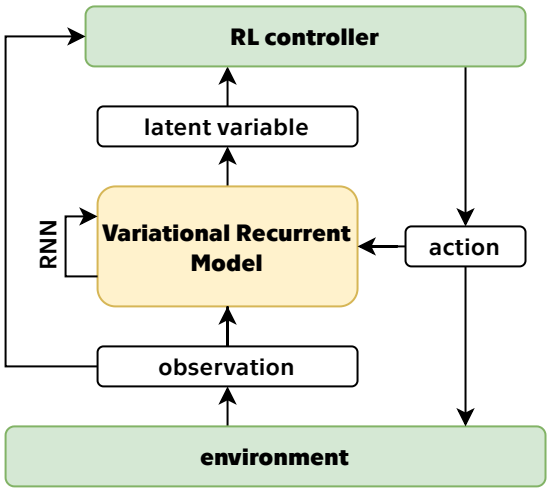


Figure 3.7: Architecture from [36] that uses a recurrent variational autoencoder to model the environment. Image reproduced from the original paper.

accuracy of the models was imperfect". Nonetheless, it was still a surprise for me, given the simplicity of this environment.

In my preliminary tests on the linearized segway environment (Chapter 4), this algorithm did not show the robustness that I had hoped for, so considering the significantly longer training time and the number of parameters possibly unsuitable for microcontrollers, I decided not to use this algorithm any further.

Chapter 4

Linearized Segway Model

The first training environment we will consider is based on a model resembling a Segway (a wheeled inverted pendulum). To be precise, we will only model the linearized dynamics to make the simulation fast and easy to implement. We will also create a full rigid body model later in 5, where we can verify the resulting controllers (and train different ones).

In Section 4.1, we go over how the linearized dynamics are obtained from the physical parameters of the Segway model. Due to reasons described in Section 4.2.1, I implemented the full procedure in Python. For completeness, I will now reiterate results from [2] and [37] that are used there. More detail can be found in the original works. Note that values of some of the parameters as well as the controller are slightly different and describe the current robot with its latest modifications. This section also includes information about the LQR controller used as a baseline throughout this chapter and also later in Chapter 6.

With the mathematical description ready, we will define the AI Gym environment that can be used to train RL controllers in Section 4.2. We will explore the effect of several hyperparameters on training of balancing (Section 4.3) and velocity tracking (Section 4.4).

The best performing agents will then be evaluated in Section 4.5 on the linear Segway model, in Section 5.5 in the rigid body simulation and finally on the real robot in Section 6.1.

4.1 Deriving the State-Space Model

We begin with a non-linear Lagrangian model for a vector of generalized coordinates $q_g = (x, \varphi, \psi)$ and input $u = (u_L, u_R)$ (see Figure 4.1) in matrix form

$$\ddot{q}_g = M(q_g)^{-1} \left(Bu - C(q_g, \dot{q}_g) - D\dot{q}_g - G(q_g) \right) \tag{4.1}$$

with inertia matrix M, Coriolis matrix C, dissipation matrix D, gravity matrix G and input matrix B. The elements of the matrices and model's parameters are given in Appendix C and Table C.1, respectively.

Because the set of second order differential equations is explicit, we can transform it into a state-space description using a standard trick by defining $q := (q_g, \dot{q}_g)$ and adding the three corresponding differential equations.

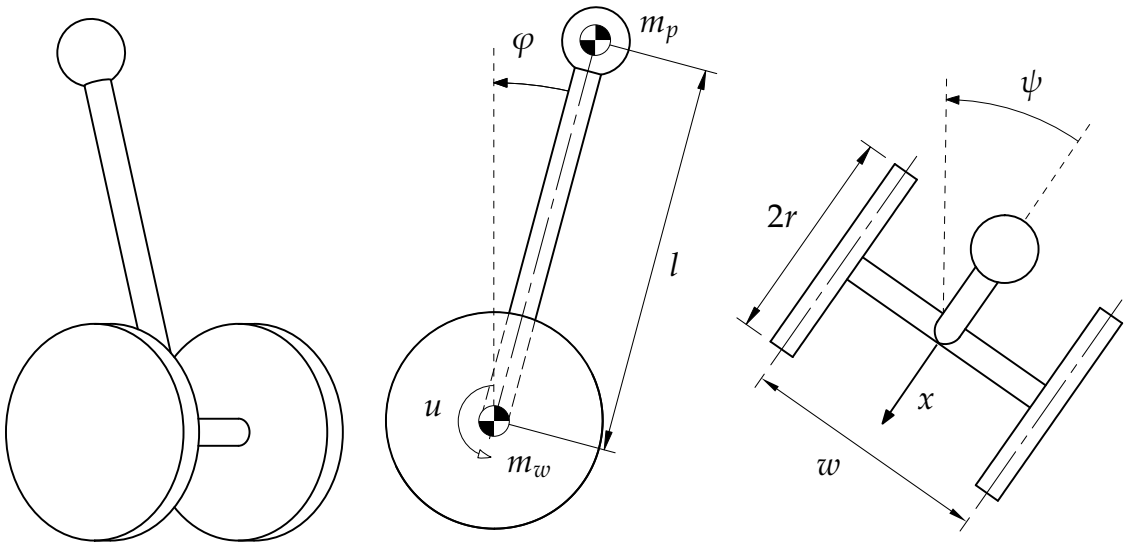


Figure 4.1: The Segway model, its state and parameters

We then find a linear approximation of the system about the equilibrium $q = \dot{q} = u = 0$, defining

$$A_c := \left.\frac{\partial f}{\partial q}\right|_{q=u=0}, \quad B_c := \left.\frac{\partial f}{\partial u}\right|_{q=u=0} \tag{4.2}$$

and then discretize it using the zero-order hold method, i.e.,

$$A_d := e^{A_c T} \quad \text{and} \quad B_d := \int_0^T e^{A_c T}\,\mathrm{d}t\, B_c. \tag{4.3}$$

The sampling period is configurable with the default value being $T = 1\,\text{ms}$ (the highest control frequency used on the real robot).

Finally, if one considers a Segway on a flat uniform surface, the pose in the plane does not affect the dynamics. Since our goal is to follow a (possibly zero) velocity reference, we can remove the decoupled states x and ψ, reducing the dimension to four and removing the non-holonomic constraints in the process. We will denote this new state by $\bar{q}$. The system matrices with the corresponding rows and columns dropped will be denoted $\bar{A}_d$ and $\bar{B}_d$. We have now arrived at our final linear discrete state-space model

$$\bar{q}_{k+1} = \bar{A}_d \bar{q}_k + \bar{B}_d u_k. \tag{4.4}$$

4.1.1 | Control

The current control algorithm which we will use as a baseline for comparison with the trained controller is based on this very model. It is an LQR running at 1 kHz. The extended system that allows for integral action of the controller (for zero steady-state error) is defined as

$$\begin{bmatrix} \Delta\bar{q}_{k+1} \\ \varepsilon_{k+1} \end{bmatrix} = \begin{bmatrix} \bar{A}_d & 0 \\ L & I_2 \end{bmatrix} \begin{bmatrix} \Delta\bar{q}_k \\ \varepsilon_k \end{bmatrix} + \begin{bmatrix} \bar{B}_d \\ 0 \end{bmatrix} u_k + \begin{bmatrix} 0 \\ r \end{bmatrix}, \tag{4.5}$$

where $r = (\dot{x}_{\text{ref}}, \dot{\psi}_{\text{ref}})$ is the velocity reference, $I_2 \in \mathbb{R}^{2\times 2}$ is the identity matrix and

$$L = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}. \tag{4.6}$$

Soon, we will need to design a reward function that will be used to train the agents. A reasonable starting point might be the criterion minimized by the LQR,

$$J = \sum_{k=0}^{\infty} \bar{q}_k^{\mathsf{T}} Q \bar{q}_k + u_k^{\mathsf{T}} R u_k, \tag{4.7}$$

which is used here to obtain the state feedback gain K that can then be used to generate inputs as

$$u_k = -K \begin{bmatrix} \Delta\bar{q}_k & \varepsilon_k \end{bmatrix}. \tag{4.8}$$

The baseline controller is generated by matrices

$$Q = \operatorname{diag}(3125, 1, 200, 1, 0.06, 0.001), \quad R = 1000 I_2. \tag{4.9}$$

For readers' convenience, I include the resulting matrix K of the baseline controller, which is

$$K = \begin{bmatrix} 3.7 & 1.5 & 3.1 \times 10^{-1} & 9.0 & -5.4 \times 10^{-3} & -8.0 \times 10^{-4} \\ 3.7 & 1.5 & -3.1 \times 10^{-1} & 9.0 & -5.4 \times 10^{-3} & 8.0 \times 10^{-4} \end{bmatrix}. \tag{4.10}$$

4.2 The Segway Environment

The de-facto standard API for training RL agents is OpenAI's Gym [38], so the simulation implemented as a part of this work conforms to it as well. The resulting `SK80_Segway` class is highly configurable: the user can define the distributions of the initial states, target states as well as specify the simulated measurement and process noise of the standard form used in LTI systems,

$$\bar{q}_{k+1} = \bar{A}_d \bar{q}_k + \bar{B}_d u_k + v_k, \quad v_k \sim \mathcal{N}(0, \tilde{Q}), \tag{4.11}$$

$$y_k = C \bar{q}_k + e_k, \quad e_k \sim \mathcal{N}(0, R), \tag{4.12}$$

using the respective covariance matrices ($C = I_4$ in our case). Note that there is a collision in standard notation and the meaning of matrices Q and R has been changed.

The default values were chosen so that the LQR is able to control the Segway model in the majority of cases. The default initial state distributions used in the experiments can be found in Table 4.1. The default process and measurement noise covariance matrices were chosen as

$$Q = R = 0.01 \operatorname{diag}(1, 5, 1, 1). \tag{4.13}$$

Additionally, the environment can visualize the situation as shown in Figure 4.2 and optionally export animations into the GIF format. It is also possible to load the environment with interactive input, where the user can control the Segway using the

	$\dot{x}_0$	$\dot{\varphi}_0$	$\dot{\psi}_0$	φ_0
distribution	$\mathcal{N}(0, 0.5)$	$\mathcal{N}(0, 0.25)$	$\mathcal{N}(0, 0.5)$	$\mathcal{N}(0, 0.25)$
clipping	-	$\lvert\dot{\varphi}_0\rvert < 0.5$	-	$\lvert\varphi_0\rvert < 0.5$

Table 4.1: The default initial states for the Segway Environment. To minimize the likelihood of uncontrollable initial conditions, the values of $\dot{\varphi}_0$ and φ_0 are clipped.

keyboard (with an underlying LQR controller for stabilization) to gauge the behavior of the system with the selected noise properties.

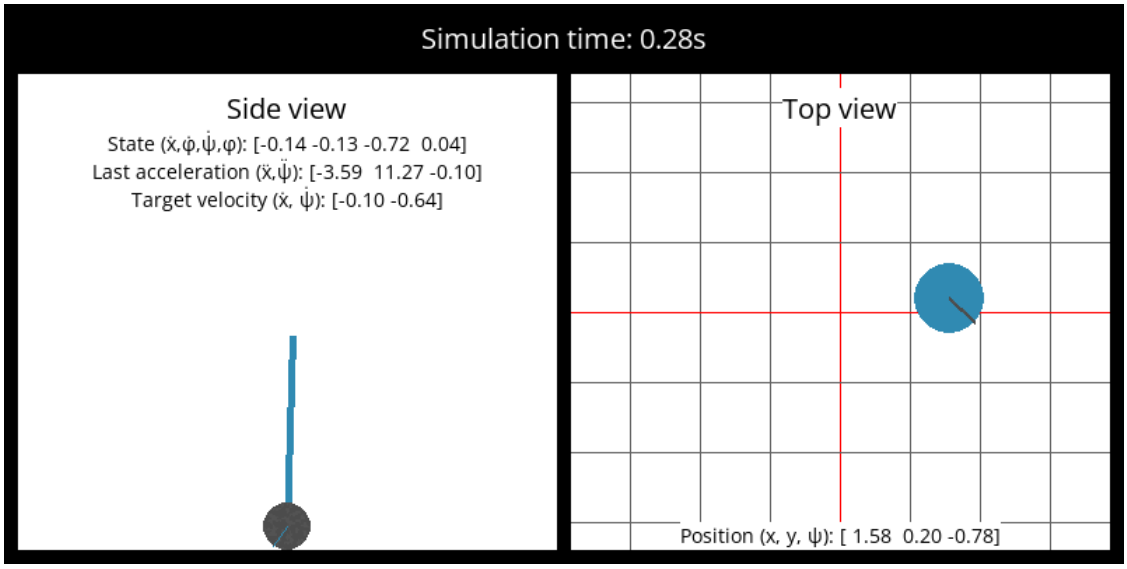


Figure 4.2: View of the linearized Segway model simulation

4.2.1 | Varying model parameters

As discussed previously in Section 3.2.4, I wanted the simulator to be able to vary the parameters of the model. Perturbing the linearized system matrices directly by a random matrix is not ideal, because the norm of the perturbation matrix is not related to the eigenvalue and eigenvector change in a straight-forward manner.

To avoid delving into pseudospectrum theory, we can illustrate this on a simple example. Consider a linear discrete system with a state-transition matrix

$$A = \begin{bmatrix} 0.9 & 10 \\ 0 & 0.9 \end{bmatrix}. \tag{4.14}$$

This system has eigenvalues $\lambda_1 = \lambda_2 = 0.9$ and so it is stable. Now we perturb the matrix by either one of two matrices

$$E_1 = 1 \times 10^{-2} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad E_2 = 1 \times 10^{-3} \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \tag{4.15}$$

with their respective norms $\|E_1\| \approx 16 \times 10^{-3}$ and $\|E_2\| = 1 \times 10^{-3}$. The eigenvalues of $A + E_1$ are $\lambda_1 = \lambda_2 = 0.91$, which is not far from the original values. Yet the eigenvalues of $A + E_2$ change much more significantly to $\lambda_1 = 1, \lambda_2 = 0.8$, despite the norm of the perturbation being over an order of magnitude smaller. Therefore, it makes sense to perturb the actual physical parameters of the model, such as mass, length, etc., and then derive the linear discrete model on-demand based on the altered parameters.

The actual implementation allows the user to specify the *percentage standard deviation* σ with the default being set to $\sigma = 10$. This argument is then used to generate a normal distribution for each model parameter p, the value of which is then sampled from a normal distribtion as

$$p \sim \mathcal{N}\left(\mu, \left(\frac{\sigma}{100}\mu\right)^2\right), \quad \text{where } \mu \text{ is the paramater value from Table C.1.} \tag{4.16}$$

Clearly, by setting $\sigma = 0$, one obtains the original model.

4.3 Balancing

Now, we are almost ready to start the learning of RL agents. The first task we will tackle is balancing: regaining the zero state from a random initial position. Because the environment introduced in the next chapter is significantly more computationally demanding (as well as harder to learn on, but let's not get ahead of ourselves), we will also test a few hyperparameter choices here with the hopes that the findings can be transferred to the non-linear simulation.

In this case, the observation space will be $\mathcal{O} = \mathbb{R}^4 \times \{0\}^2$ describing the reduced state $\bar{q}$ and the zero velocity reference vector (for compatibility with the models trained in Section 4.4). The action space will be $\mathcal{A} = [-1, 1]^2$, describing the torques applied to the left and right wheel respectively. The episodes are truncated after 20 s to let the agent experience new initial conditions even after it learns to stabilize the Segway.

4.3.1 Rewards

There are three parts to an environment that we have discussed in Section 3.1. Equation (4.11) gives us function τ that describes the dynamics and Equation (4.12) defines function ω that generates the observations. What is left to define is the reward function ρ, so let us focus on that for a moment.

Designing a (good) reward function is not a trivial task. Generally, researchers present their results on already predetermined (set of) environments, such as the Atari, DMControl or the AI Gym benchmarks where rewards are set in stone. Although some theoretical results have been achieved, reward function design is a largely empirical process guided by domain expertise [24].

Luckily, we can get inspired by the LQR criterion from Equation (4.7) and design a reward function that penalizes nonzero states, as well as system input and reference error magnitude. Additionally, I decided to include a penalty for falling, which is in this case defined as $|\varphi| > 1$ rad and a reward for each step, which is meant to compensate for degenerate strategies where falling immediately might obtain better return than poor

name	symbol	value in evaluation	training search		
			distribution	range	best
reference error	c_ε	50	log uniform	[0.1, 10]	8
wheel input	c_w	1	log uniform	$[1 \times 10^{-4}, 1]$	0.2
step cost	c_s	0	uniform	[-1, 0]	-0.4
pitch angle	c_φ	1	uniform	[0, 1]	0.7
fall penalty	c_f	10000	-	-	100
error power	p	2	-	-	1

Table 4.2: Parameters of the rewards sweep for Segway balancing

balancing. The reward at time t is then given as a linear combination

$$\rho_{\text{balance}}(\bar{q}, r, u) = -c_\varphi \varphi^2 - c_\varepsilon \|\varepsilon\|^p - c_w \|u_w\|^p - c_s - c_f \text{fallen}(\varphi), \tag{4.17}$$

where ε is the reference error is defined as $\varepsilon := (\dot{x} - \dot{x}_{\text{ref}}, \dot{\psi} - \dot{\psi}_{\text{ref}})$, $u_w = (u_L, u_R)$ is the action input to both wheels and the last term is defined as

$$\text{fallen}(\varphi) = \begin{cases} 0 & \text{if} \quad |\varphi| \leq 1, \\ 1 & \text{if} \quad |\varphi| > 1. \end{cases} \tag{4.18}$$

Each of the terms is additionally normalized to a range of $[-1, 1]$ before the coefficient is applied to make the coefficients comparable. For example, the maximum considered reference error was $\varepsilon_{\max} = (2, 4)$ and its norm was used as the scaling coefficient.

To find a reasonable combination of coefficients, I used the SAC algorithm with parameters as suggested in [7]. Unless otherwise specified, this will be true for all training runs presented here. Furthermore, I used what I will refer to as *ideal conditions*, where the model is not randomized and there is no noise present in the system, in an effort to make the training faster.

After I set $c_f = 100$ to anchor the coefficients somehow, I ran a Bayesian search (as a W&B sweep[1]) to maximize the mean episodic reward in evaluation. The coefficients for evaluation were chosen so that it is clearly visible whether the agent has fallen and the power p of the errors was set to $p = 2$ with the intention to make large errors more visible at first glance. On the other hand, during training, it was set to $p = 1$. The reason for this will become clear in Section 4.4. The evaluation coefficients, search ranges and results of the sweep are shown in Table 4.2.

4.3.2 | Experiments

During reward search, virtually all evaluation runs ended in a timeout already after 20k environment interactions (recall Figure 3.1 that illustrates a single interaction) and the top performer has not fallen a single time after 100k interactions. Granted, no noise nor model uncertainty was present but nonetheless, this was a promising start. It would

[1] https://docs.wandb.ai/guides/sweeps

	number of hidden layers			
number of neurons per layer	1	2	3	4
64	578	4738	8988	13058
128	1154	17666	34178	50690
256	2306	68098	133890	199682

Table 4.3: Number of parameters of an MLP with 6 inputs and 2 outputs, as used for Segway stabilization

seem that this task is even a little too simple and that changing the hyperparameters will not have much effect. Therefore, we will limit ourselves to just three experiments here.

If the default SAC networks can "solve" the problem so fast, it might be interesting to see just how small the networks can be before we run into issues even in the ideal conditions. After all, the controller will be deployed on an embedded device running on a battery, so model complexity is a significant factor. The results of this experiments are shown in Figure 4.3.

Note that in all the experiments in this chapter, each run was evaluated five times with different seeds for the pseudorandom number generator. The shaded area then corresponds to the range of measured values and the line is their mean. The value at each point is the average of 50 evaluation episodes. If applicable, the baseline value was computed on 1000 evaluation episodes with the same settings, controlled by the LQR from Equation (4.9).

At their best, all architectures perform similarly. However, when there is only a single hidden layer or when the two hidden layers are small (64 neurons), the agent appears to have fallen during some evaluation episodes. For reference, you can check the number of parameters of each network, which should be roughly proportional to computational demands, in Table 4.3.

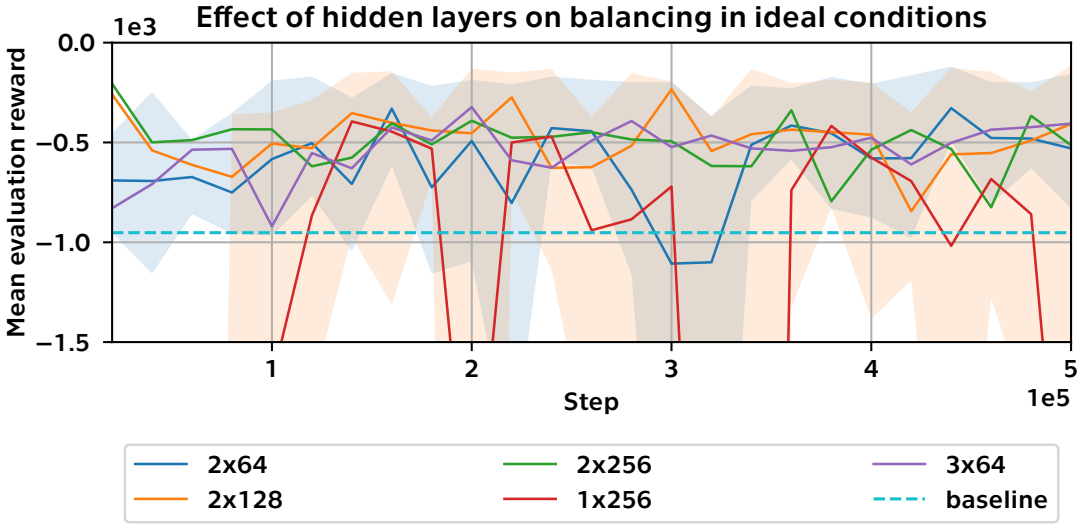


Figure 4.3: The two smallest networks are sometimes outperformed by the baseline controller. Note that only the ranges of the two are displayed to improve readability.

For the next experiment, we depart from the ideal conditions and look at how *frame stacking* – using last n observations as the input – helps the performance in the presence of model uncertainty and noise. The answer, as demonstrated by Figure 4.4, is that it makes a significant difference. However, increasing the number of observations from $n = 4$ to $n = 10$ did not improve the episodic rewards any further.

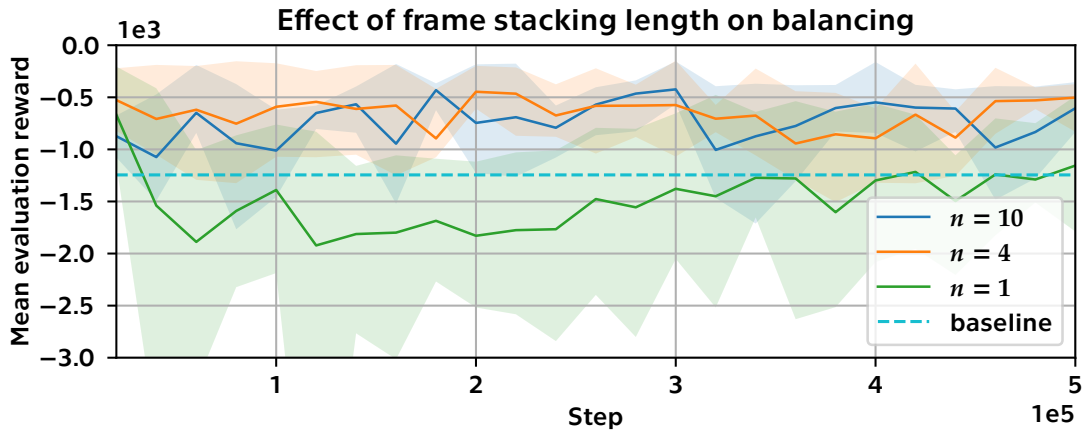


Figure 4.4: Stacking several last observations for the model input significantly improves performance in the presence of noise.

Lastly, I tested the three architectures that performed well in the first experiment with frame stacking to see if we can further remove some architectures from our considerations. It turns out that the choice out of the configurations tested does not have much of an effect. In an effort to make the text more cohesive, the plots from such experiments will be included in the appendix. You can find the corresponding figure in this case in F.1.

4.4 Velocity Tracking

Balancing the robot on the spot was successful, so what about a more general task? Instead of requesting the zero state, we choose a specific reference $r = (\dot{x}_{\text{ref}}, \dot{\psi}_{\text{ref}}) \in [-1, 1] \times [-2, 2]$ for each episode. As before, I started with a sweep on the rewards.

4.4.1 Rewards

The reward function for velocity reference is almost identical to Equation (4.17) except there is an additional term that grants an additional reward when the agent reaches the reference velocity. But what constitutes reaching the goal? If we simply check whether the state is close to the target state, we might incentivize the agent to disregard anything that comes afterwards. This would likely cause the controller to overshoot the targets in practice on a good day and often cause it to fall moments after reaching the target (which would not happen during training). Clearly, we care about acceleration (or rather, the velocity difference, since we are dealing with a discrete system) too.

The first solution I tested was simple – putting an additional condition on the magnitude of the acceleration a_t. This works fine in ideal conditions but the presence of noise (especially process noise), the situation is more problematic. With increasing process

name	symbol	value in evaluation	training search		
			distribution	range	best
reference error	c_e	50	log uniform	[0.1, 10]	0.7
wheel input	c_u	1	log uniform	$[1 \times 10^{-4}, 1]$	1.6×10^{-3}
step cost	c_s	1	uniform	[0, 1]	0 (-0.4)
pitch angle	c_φ	0	uniform	[0, 1]	0.7
fall penalty	c_f	10000	-	-	100
error power	p	2	-	-	1

Table 4.4: Parameters of the rewards sweep for Segway velocity tracking

noise, the respecitve component of $\|a_t\|$ rises as well. Hence this approach would likely require variable goal definition based on the amount of process and measurement noise.

Instead, I opted for a different solution. The environment keeps track of the exponential moving average of the acceleration vector,

$$\bar{a}_t := \alpha a_t + (1 - \alpha)\bar{a}_{t-1}, \tag{4.19}$$

and tests for the magnitude of $\|\bar{a}_t\|$. From a theoretical standpoint, this solution is not ideal, because it breaks the Markovian property (or rather, makes the process only partially observable). Nonetheless, as we will see, this is fine in practice.

The final reward function is

$$\rho_{\text{velocity}}(\bar{q}, r, u, \bar{a}_t) = \rho_{\text{balance}}(\bar{q}, r, u) + c_f \text{success}(\bar{q} - r, \bar{a}_t), \tag{4.20}$$

where

$$\text{success}(\varepsilon, \bar{a}_t) = \begin{cases} 1 & \text{if} \quad \|\varepsilon\| < 0.1 \quad \text{and} \quad \|\bar{a}_t\| < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4.21}$$

Note that the coefficient used with *success* is the same as the one used for falling, only the term has an opposite sign.

Unfortunately, using the coefficients found in the stabilization task causes the agent to learn the locally optimal strategy, in which the agent learns to fall as soon as possible. I suspect that the issue lies in the fact that the initial reference error is typically much larger here than in the case of zero reference.

Whatever the reason may be, I ran the sweep again to find a new set of coefficients. Its results can be found in Table 4.4. Note that the error coefficient is a magnitude lower, which supports the theory about the agent falling on purpose.

After the sweep was finalized, I made one additional experiment regarding rewards. The resulting step coefficient c_s was rather close to zero, which motivated me to see whether we can do without it altogether. The issue with positive c_s is that receiving a reward for each step might incentivize the agent to prolong the episodes, for example by staying close to the reference to minimize the error penalty but not close enough as to reach the goal and end the run.

Hence I performed an experiment in which I vary the step reward and keep everything the same. It turns out that setting $c_s = 0$ shows similar performance and lower variance

(as seen in Figure F.2) than the value found by the sweep. Therefore, I adjusted its value to $c_s = 0$, because the term is not necessary to counter the undesirable "immediate fall" strategy and may potentially cause other issues.

Intrigued by the outcome, I ran another experiment, this time with varying c_ε. However, I did not made any adjustments based on the results in Figure F.3.

4.4.2 | Choices of hyperparameters

Following a reference appears to be noticeably more difficult for the agent than just stabilizing the Segway to the extent where the baseline LQR controller typically outperforms it, despite not being optimized for precisely this form of reward. We should, therefore, have an easier time discerning the effects of different hyperparameters.

It is time to address the value of p. Why not just use quadratic errors that are common in control and easier to both compute and differentiate? Well, it turns out that $p = 1$ gives better results, as shown in Figure 4.5. Recall that in evaluation, we use $p = 2$, so if anything, this metric is biased towards $p = 2$.[2] I am unsure about the reasons behind this phenomenon but during our meeting with prof. Robert Babuška[3], he affirmed that this was his experience as well. Therefore, I will henceforth only use $p = 1$ during training.

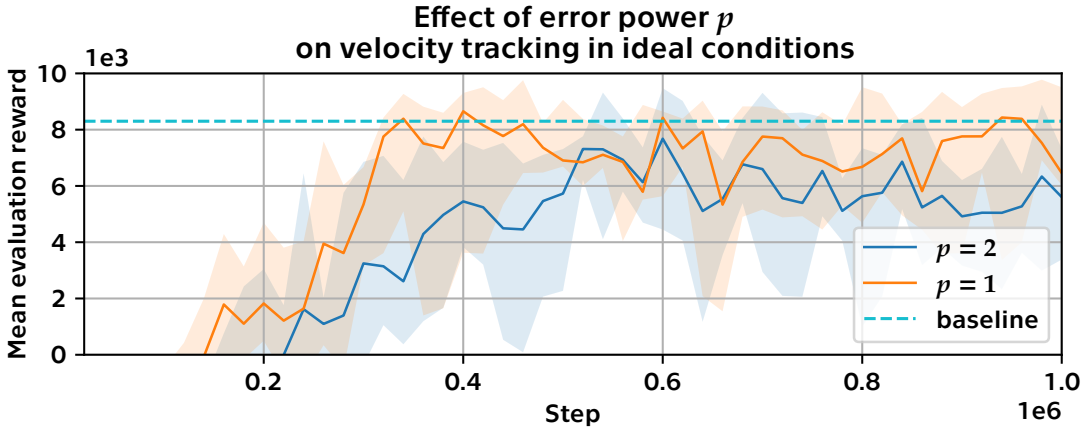


Figure 4.5: Absolute errors are more beneficial to learning than quadratic errors.

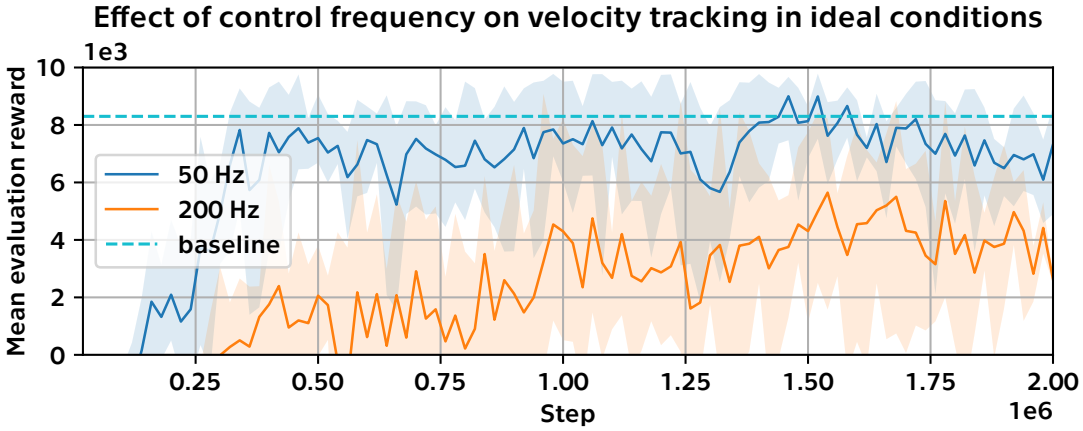
Another experiment that may surprise a control engineer concerns control frequency. Intuitively, the agent should be better posed to control the system when the period between actions is lower. However, the results in Figure 4.6a, where we compare control frequency at 200 Hz vs 50 Hz tell a different story. Note that the rewards in evaluation are proportionally scaled, except for the one-time rewards for falling and achieving the goal, to make the numbers comparable.

My explanation for this observation is as that for each observation the slower agent sees, the faster one sees four. However, these are very similar to one another and as such do not add much extra information to the replay buffer and perhaps even cause overfitting. Additionally, for the same amount of interactions, the slower agent experiences four times as many episodes.

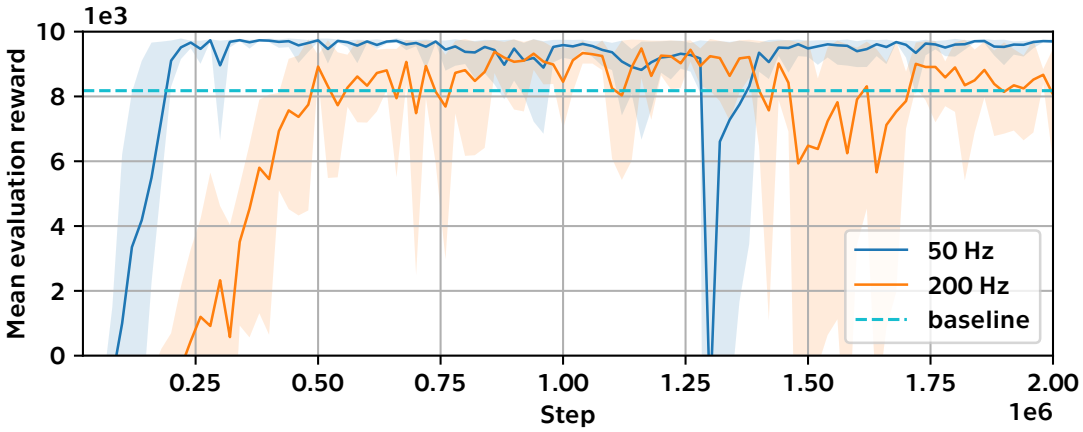
[2]Note that the coefficient c_s was reoptimized for the quadratic error to avoid bias.
[3]Professor of intelligent control and robotics at TU Delft, a researcher in reinforcement learning.

If my explanation holds, adding process and measurement noise might help. Therefore, I ran the experiments again, this time with noise and frame stacking with $n = 4$. If the issues are due to overfitting, they should be mitigated by the noise to some extent.

The results support my theory – the network trained on faster control frequency still takes longer to train (which can be explained by the fact that it takes longer to generate meaningfully different trajectories) and the performance is still somewhat lacking, yet the difference in Figure 4.6b is not as pronounced.



(a) In ideal conditions, the slower controller learns a significantly better policy.



(b) In noisy conditions, the faster controller almost catches up (with higher variance).

Figure 4.6: Comparing the learning of 50 Hz and 200 Hz SAC controllers.

4.4.3 | Frame stacking

With our previous choices justified, we can mostly repeat the experiments from balancing. The results of frame stacking in Figure F.4 are largely the same as in the case of stabilization. Curiously, the performance in a noisy environment actually surprasses the results achieved in *ideal conditions*. I suspect that this may be due to the same reason the 200 Hz controller was improved in such a case – mitigation of overfitting.

Additionally, I was curious to see whether adding the past actions to the input along with past observations would improve performance – theoretically, knowing the past actions may help the agent to identify the model to some extent and correct for it. However, that does not seem to be the case, as shown by Figure 4.7. It is possible that using a longer observation window (possibly strided to decrease input dimension) could lead to this effect but I leave that as an open question.

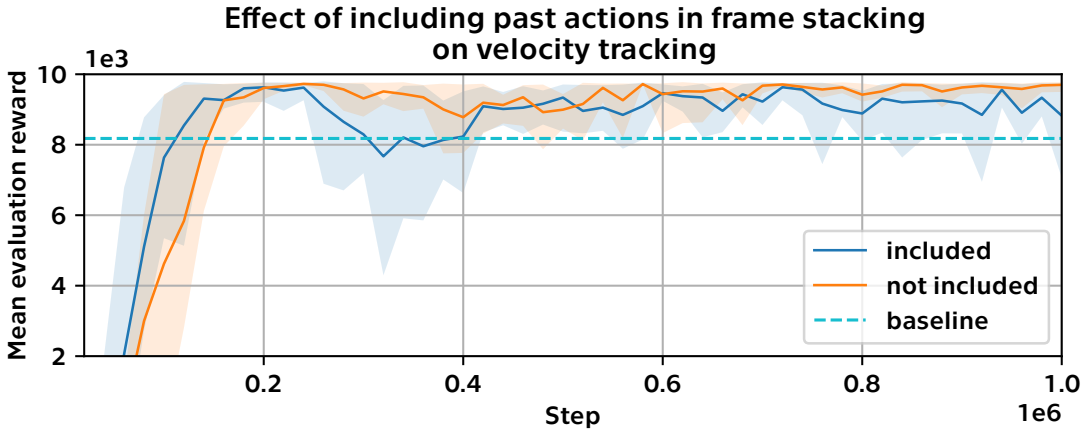


Figure 4.7: Including past n actions along with the observations decreased the performance of frame stacking.

Finally, I tested the effect of network structure and again found that it largely did not matter (the results are shown in Figure F.5). With the idea that perhaps a more significant change in architecture was needed, I decided to see whether using the D2RL algorithm brings any benefits to the table. As it turns out, SAC learns faster and achieves better and more stable performance, as demonstrated in Figure 4.8.

4.5 | Verification

In the previous section, we have seen more than a hundred models that were trained on the linear model introduced in this chapter. Although the evaluation results tell a part of the story, the reward function used for comparison was still somewhat arbitrary. It is here that we will finally get to see how well the agents actually perform on the linear model. Experiments on the MuJoCo model and real robot will be shown later in sections 5.5 and 6.1.

I will only show one experiment per category in full (that is, with all the state variables). In other cases, only the variable(s) of interest will be shown. This was motivated by the desire to keep the document brief.

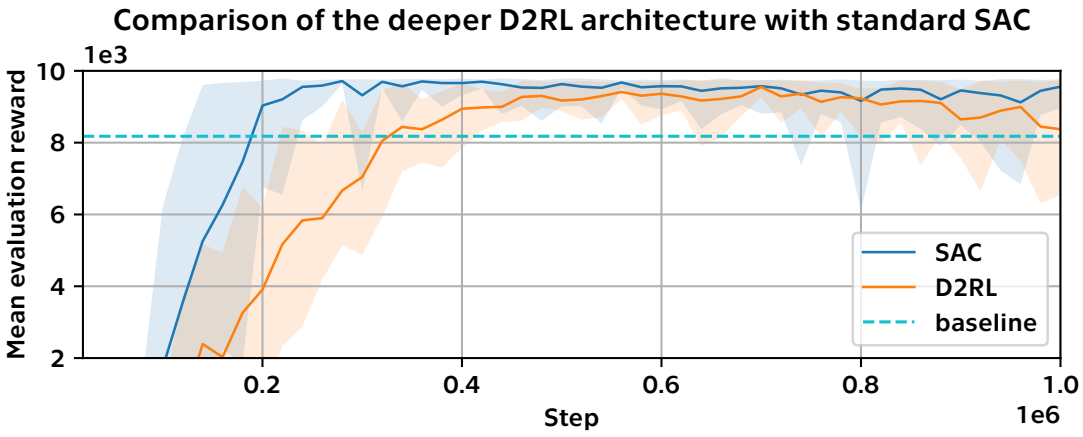


Figure 4.8: SAC with default settings outperforms D2RL when trained on the Segway model for velocity reference tracking.

When it comes to balancing, we start by comparing the model that was trained in ideal conditions to the one that experienced a more diverse environment. Interestingly, while the agent that has not experienced noise before does tend to fall more often (depending on the initial configuration), when it finds itself around the equilibrium, it is surprisingly comparable to the controller trained with noise, as illustrated by Figure 4.9.

Figure 4.9: In some cases, the agent trained in ideal conditions manages to compete with the one that experienced noise during training.

Another observation we can make is the fact that the controller trained for reference tracking manages to stabilize the Segway perhaps even better than the agent that specializes on this task. This is true to such extent that one may wonder whether it even makes sense to train specialized agents for stabilization, besides for debugging reasons.[4]

Finally, one run with all the state variables is shown in Figure 4.11.

[4]Yes, this is foreshadowing of the results on the real robot, found in Section 6.1.

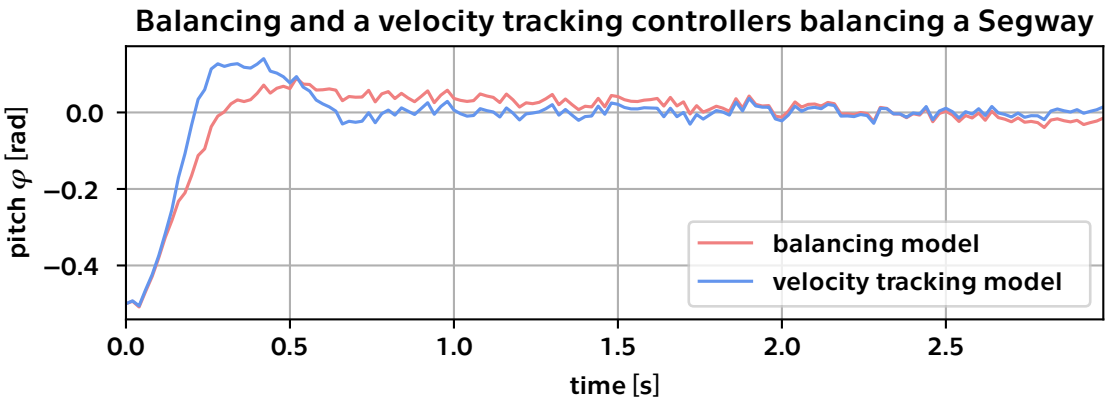


Figure 4.10: The model trained for velocity tracking performs essentially the same.

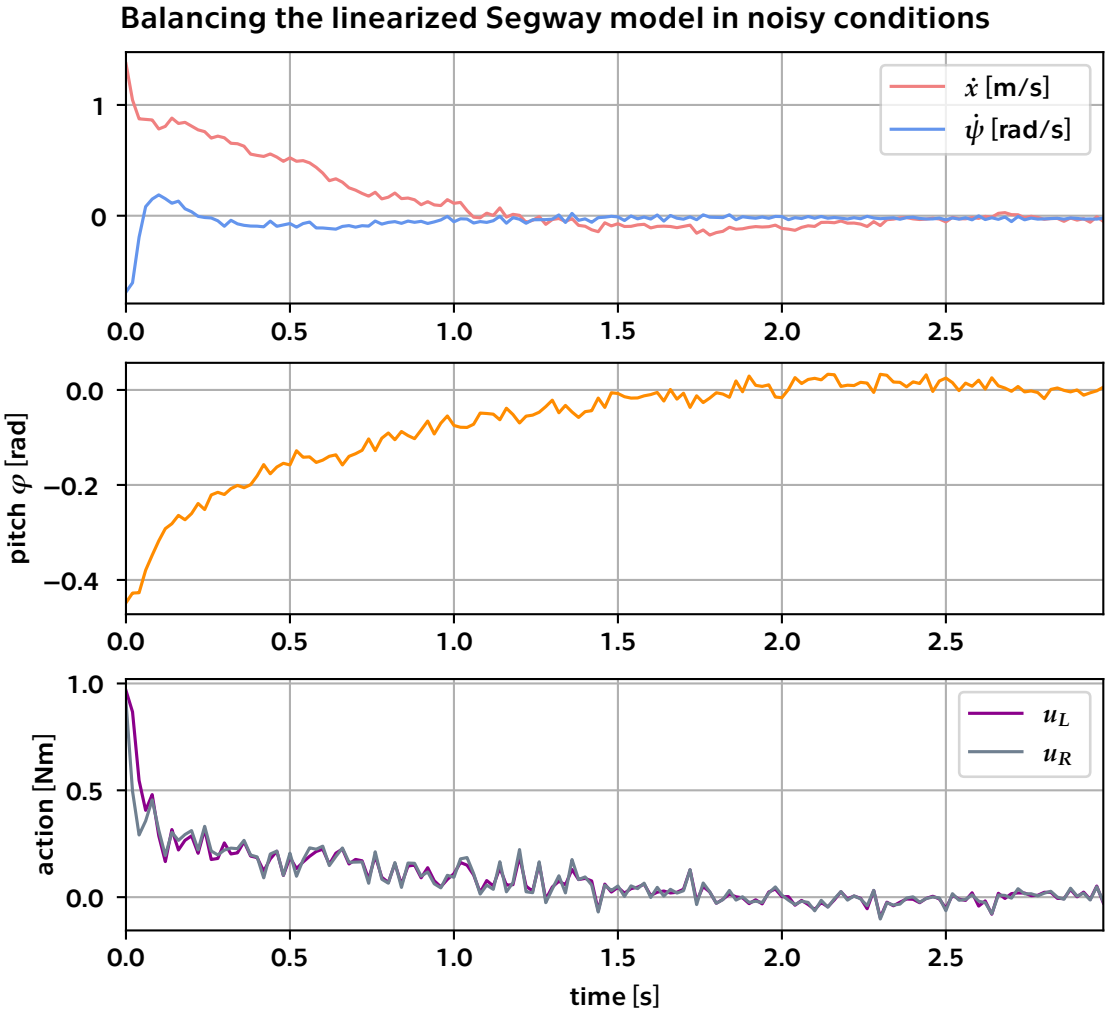


Figure 4.11: A balancing run with all the states plotted

4.5.1 Velocity tracking

The impact of noise in training is much more severe when we use nonzero reference, as illustrated by Figure 4.12. On the other hand, a change in experiment setting that does not cause any issues is, almost ironically, increasing the control frequency to 200 Hz, which can be found in Figure 4.13.

Figure 4.12: In the case of velocity tracking, training with noise makes a notable difference when noise is present.

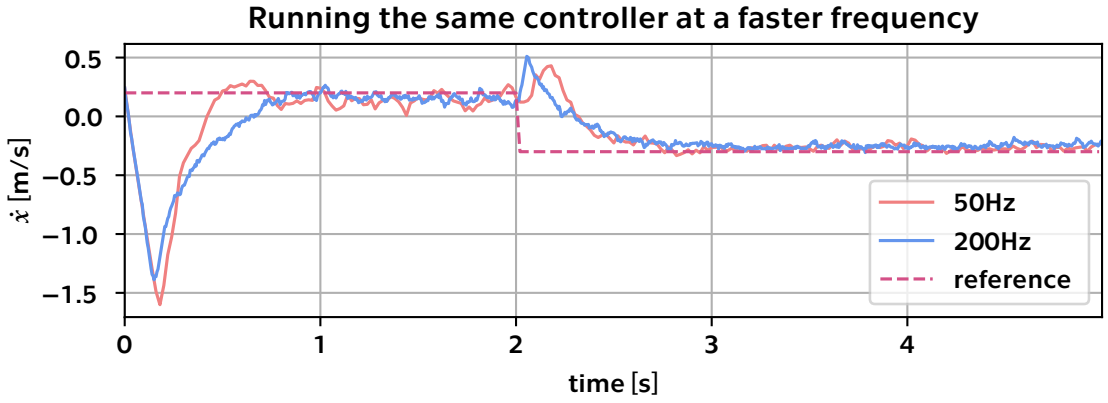


Figure 4.13: The model is not significantly affected by faster control frequency.

For reference, in Figure 4.14, I compare the best performing agent also with the LQR running at 1 kHz. Please note that the controller was tuned for the actual robot and not for the linear model, hence its response may seem slightly worse than it is in practice. Finally, a full example can be seen in Figure 4.15.

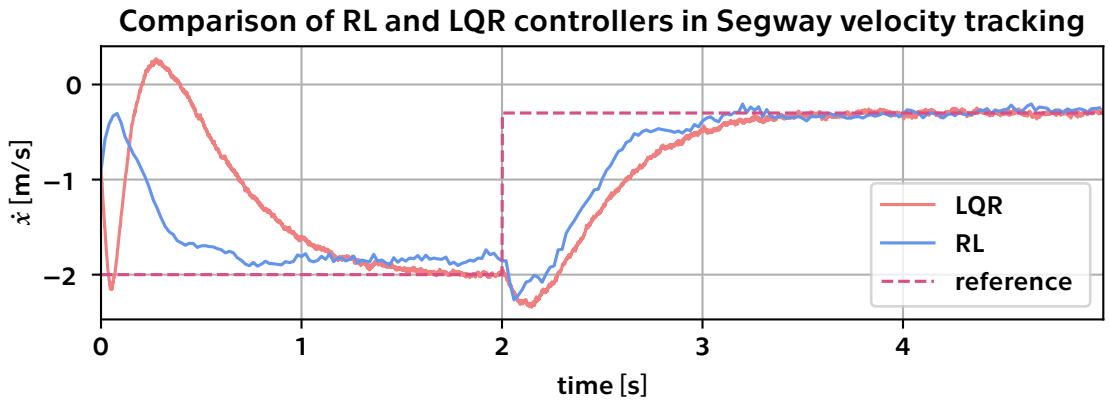


Figure 4.14: The performance is comparable to the LQR from Equation (4.9). Notice that the RL managed to get close to $\dot{x}_{\text{ref}}$, despite never seeing reference higher than $\dot{x}_{\text{ref}} = 1$ during training.

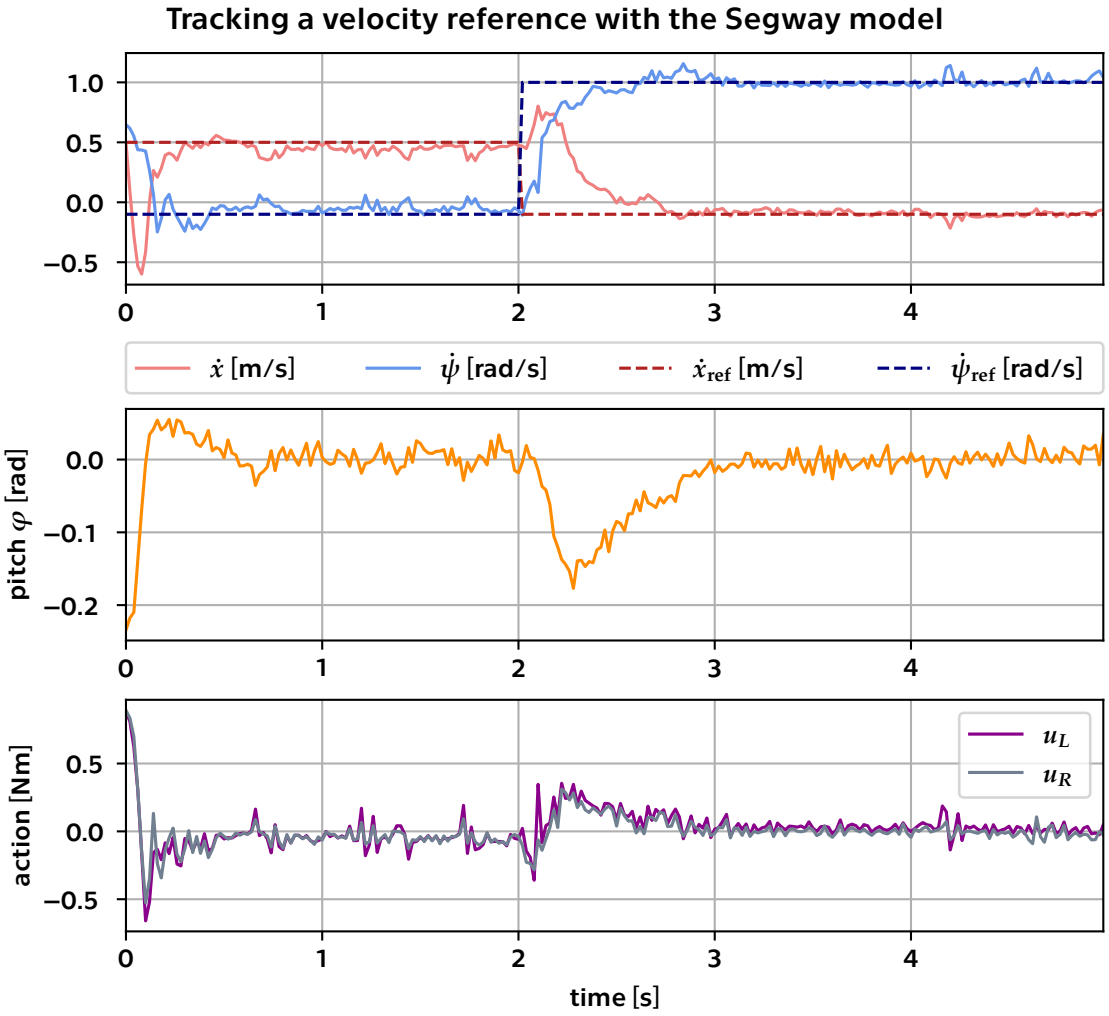


Figure 4.15: A velocity tracking run with all the states plotted

Chapter 5

Rigid Body Model

The second model and training environment we will introduce uses a full rigid body simulator of the robot. As the robot was designed in-house, we have the original 3D files at our disposal. These provide us with accurate dimensions of the robot and they can also be used to infer (simplified) collision geometries.

As a part of his diploma thesis, Adam Kollarčík also created a full simulation of the robot in Gazebo and Simulink [2]. Gazebo is a simulation framework commonly used in robotics, being particularly useful when testing control strategies on the Robot Operating System (ROS). However, the SK8O robot does not run ROS, so that is not an advantage for us.

For our use-case, it would have been ideal if it were possible to easily interact with the gazebo simulation to make it compatible with AI Gym API. Unfortunately, it seems that gazebo simulation is not very popular among RL researchers. There are multiple projects aiming to make gazebo seamlessly available in Python for RL, such as OpenAI ROS, gym-gazebo2, DeepSim, FRobs_RL or gym-ignition. However, they have all been left unmaintained for over a year or downright archived for one reason or another.

A possible reason why there is no simple/standard workflow for gazebo in RL might be the popularity of a "rival" framework MuJoCo [39]. This simulation environment is the de-facto standard for benchmarking RL algorithms on 3D problems and is used in most of the research papers that deal with continous control referenced in this thesis.

This framework used to be prohibitively expensive (3000$ per year for a single lab license), but was acquired by DeepMind in 2021 and made open-source in 2022 [40]. Naturally, I was curious to use this software now that it is available. It supports everything necessary for our use-case, including simulation of closed kinematic chains, which is one of the most challenging apects of SK8O that many frameworks do not support.

MuJoCo's own format, MJCF, has a fundamentally different approach to model definition than gazebo's SDF, in which the previous model was implemented. While they both support the URDF format, it does not support kinematic loops, so chaining the conversions as SDF → URDF → MJCF is not possible.

There is a tool for converting gazebo models into MuJoCo models.[1] However, in early 2023, it was still a work-in-progress and it also did not support closed kinematic chains. After many attempts to do the (partial) conversion, I gave up and decided to recreate the model essentially from scratch.

Identification of the parameters was not a part of this thesis and is not my work, with one minor exception which we will get to later. I compiled the parameters from Adam's thesis, his code and other sources from the team behind SK8O and present them here, in one place, for future reference.

5.1 MuJoCo Simulation

Unlike in the linear case, there is not much to be said about the physics, since almost everything is being done behind the scenes by the simulation framework. We will mostly deal with the high level structure of the robot and its parameters.

You can see approximately how the body is constructed in Figure 5.1, though I leave the concrete values of all the parameters for Appendix D. Note that all the Centers of Mass (COMs) are placed in the geometrical centers of the links, each of which can be roughly approximated by a box, unless otherwise specified. I took special care to position all the joint coordinate system origins on each side of the robot in a single plane and to set their zero points and directionalities to simplify algebra in Section 5.2.1.

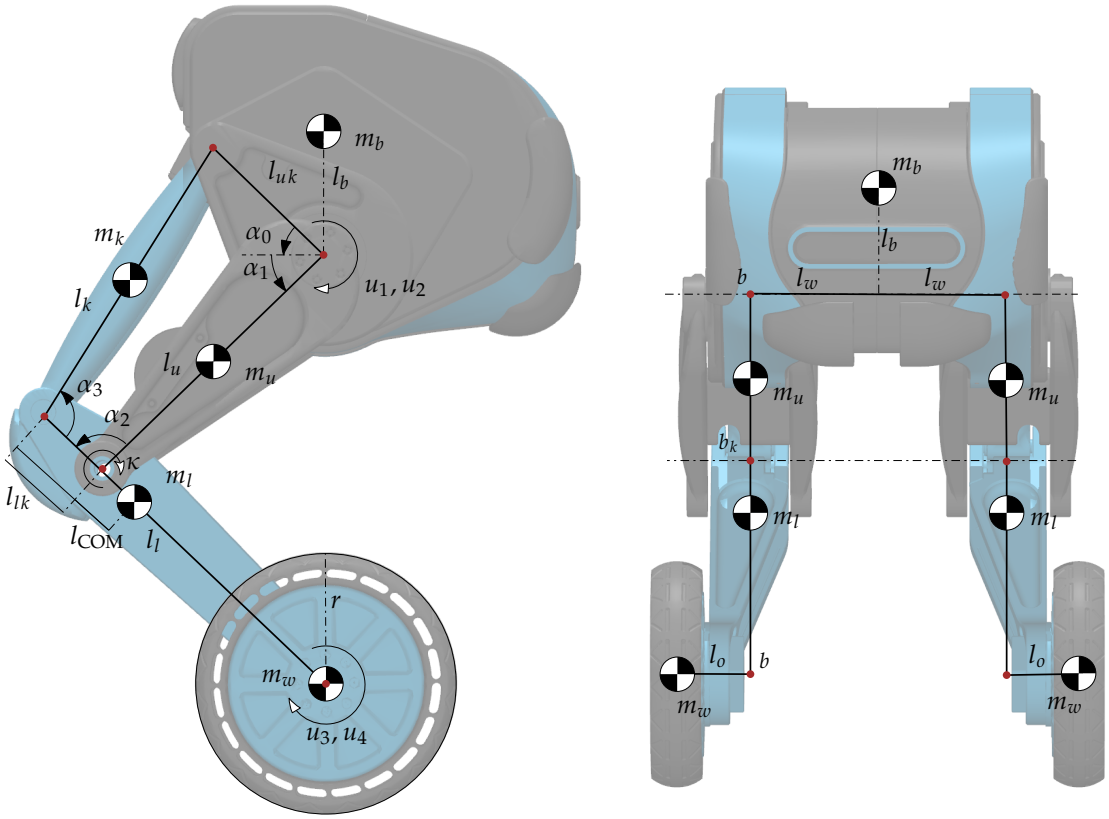


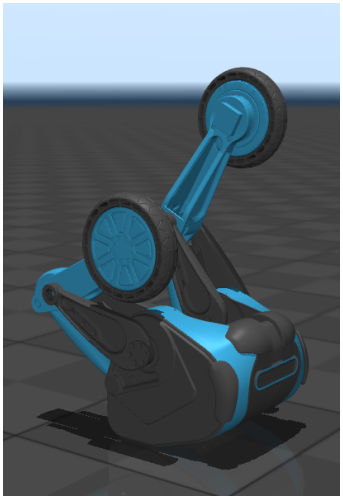
Figure 5.1: Diagram of SK8O's structure superimposed over the 3D mesh

[1] https://github.com/gazebosim/gz-mujoco/tree/main/sdformat_mjcf#tools-for-converting-sdformat-to-mjcf

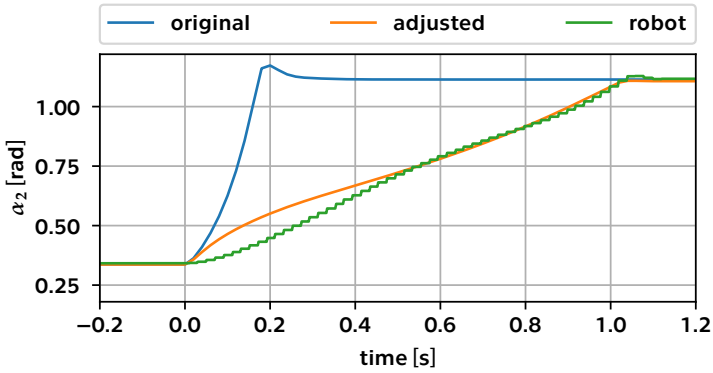
There are two properties of the robot that need special deliberation. First, there are torsion springs at the knees, each of which was modeled to generate torque

$$\tau = -\kappa(\alpha_{2,\text{ref}} + \alpha_2) \tag{5.1}$$

in the direction that counters the weight of the robot's body in the upright pose. I adjusted the value of several joint properties compared to [2] for a more accurate description of reality. The goal was to match the joint angle trajectories when extending a leg when the robot is placed upside down (as shown in Figure 5.2). We also confirmed that the time it takes for the robot to fall when there are no inputs at the hip motors is comparable in simulation with the adjusted parameters and on the real robot.



(a) Render of the experiment



(b) Comparison of the original and modified parameters, along with the measured values of a leg extension. The overshoot with original parameters is due to excessive force being developed against the soft constraint that models the kinematic loop.

Figure 5.2: Justification for alterning the knee damping coefficient

The second parameter I had to adjust was the friction between the wheels and the floor, because it is modeled differently in gazebo. I set the tangential, torsional and rolling coefficients[2] $f_{\text{tan}}, f_{\text{tor}}, f_r$ as described in Table D.1 by visually comparing the behavior of the simulation and the real robot, making sure that slippage is unlikely in the simulation and that the robot can rotate around one of the legs. Because we assume no slippage in normal operation, I believe this to be an accurate enough assessment.

The joint configuration I embedded in the model XML file was found by leaving the LQR to stabilize the robot and capturing the final position. You can view the angles in Table 5.1. The resulting model can be seen in Figure 5.3.

angle	α_0	α_1	α_2	α_3
limits [deg]	[45, 45]	[20, 64]	[51, 143]	-
value in XML [deg]	45	43.8	92.4	100.5

Table 5.1: A joint configuration for a stable position of the robot, along with joint limits imposed by the physical structure of the robot. The limits are set slightly smaller than on the real robot to allow for some leeway in the solver.

[2] https://mujoco.readthedocs.io/en/stable/modeling.html#contact-parameters

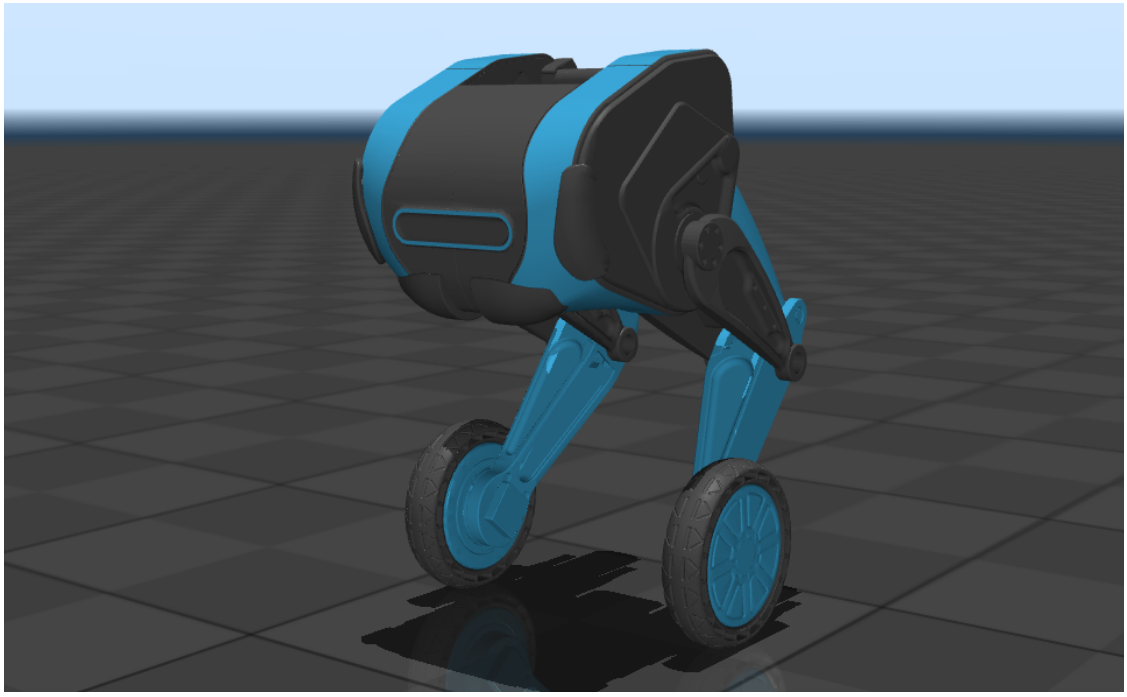


Figure 5.3: Resulting 3D model in MuJoCo

5.2 The MuJoCo Environment

The simulation is included in an AI gym compatible class, `SK8O_Full`. The environment is highly configurable regarding initial conditions, noise properties etc., just like the linear one. Additionally, it supports 3D rendering of the episode, which is (this time) implemented directly by the parent class in the Gym library.

Measurement errors are simulated in the same way as in `SK8O_Segway`. On the other hand, the process noise is modeled very differently – I apply random forces and torques to the COM of the robot. Of course, such noise does not account for forces constant in time, such as someone pushing the robot or a slightly different position of the COM. However, this issue should be solved by domain randomization.

In addition to randomizing the masses, moments of inertia, damping coefficients, etc., I also vary the positions of the COM of each of the bodies at the start of each episode. To randomize these positions, we cannot sample from a normal distribution centered at the true value with a proportional standard deviation, like we did in Equation (4.16). This is because the result would depend on the choice of origin – for example if the COM is placed at the origin, there would be no variability. Instead, I sample the position in each axis with the standard deviation being proportional to the size of the link in that axis (recall that all of the SK8O's body parts can be reasonably approximated by a box). Offseting the COMs should have the same effect as constant forces.

5.2.1 Initial configurations

So far, we have defined an XML file that describes the robot in one of many stable positions. However, any reasonable implementation of the simulator should allow the user to choose any (physically possible) starting configuration they wish. A certain amount of randomness in the initial position is also key for proper training of the NNs, as we have discussed in Section 3.2.4.

Of course, MuJoCo is "just" a general physical simulator and as such does not provide inverse kinematics. The problem is compounded by the fact that the topology (closed kinemtic loop) of each leg dictates that its configuration is fully determined by any of the respective joint coordinates (barring wheel rotation). For users' convenience, I implemented two possible initialization methods:

- setting α_1 – the angle(s) at the hip,
- setting h – the height of the equivalent Segway model.

Below, I go over the algebra necessary to compute all the joint coordinates depending on the method chosen. Admittedly, this is nothing but high-school trigonometry. However, it does take a considerable amount of time and drawings of triangles to get to the correct solution and therefore, I will go over it here with the hopes that no one will have to solve this problem again in a year's time.

The approach presented below uses the law of cosines repeatedly. In general, this approach is problematic due to cosine not being injective (or, equivalently, due to its inverse having range $[0, \pi]$), which may not reveal all possible solutions. However, this is not a problem in our case due to the kinematic constraints – in fact, it is an advantage, because it implicitly discards the invalid solutions and it is also the motivation behind its use.

Hip angle initialization

In this scenario, we know the angle at the hip, α_1 (and α_0, which is fixed). We can than find one of the diagonals in the kinematic loop, depicted as d_1 in Figure 5.4, using

$$d_1 = \sqrt{l_u^2 + l_{uk}^2 - 2 l_u l_{uk} \cos(\alpha_0 + \alpha_1)}. \tag{5.2}$$

This in turn can be used to compute the two partial angles of $\alpha_2 = \alpha_{2a} + \alpha_{2b}$ by applying the law of cosines again, as

$$\cos \alpha_{2a} = \frac{d_1^2 + l_u^2 - l_{uk}^2}{2 l_u d_1}, \tag{5.3}$$

$$\cos \alpha_{2b} = \frac{d_1^2 + l_{lk}^2 - l_k^2}{2 l_{lk} d_1} \tag{5.4}$$

and then taking the inverse.

When we know that, we can find last missing angle by solving for

$$\cos \alpha_3 = \frac{l_{lk}^2 + l_k^2 - d_1^2}{2 l_{lk} l_k}. \tag{5.5}$$

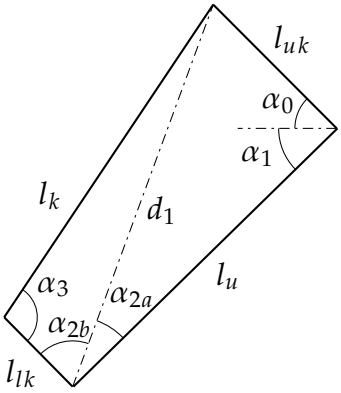


Figure 5.4: A drawing of the relevant quantities used in hip angle initialization

Leg length initialization

In this case, we know the height of the robot depicted as h in Figure 5.5. The solution is very similar to the one above. We compute α_4 by

$$\cos\alpha_4 = \frac{h^2 + l_l^2 - l_u^2}{2l_l h}, \tag{5.6}$$

which is then used to compute the length of the second diagonal of the kinematic loop,

$$d_2 = \sqrt{(l_l + l_{lk})^2 + h^2 - 2h(l_l + l_{lk})\cos\alpha_4}. \tag{5.7}$$

This diagonal can then be used to solve for the partial angles of α_1 using

$$\cos(\alpha_0 + \alpha_{1a}) = \frac{l_{uk}^2 + d_2^2 - l_k^2}{2d_2 l_{uk}}, \tag{5.8}$$

$$\cos\alpha_{1b} = \frac{d_2^2 + l_u^2 - l_{lk}^2}{2d_2 l_u}. \tag{5.9}$$

From then on, we can use the procedure above to get the all of the necessary angles to set the desired configuration.

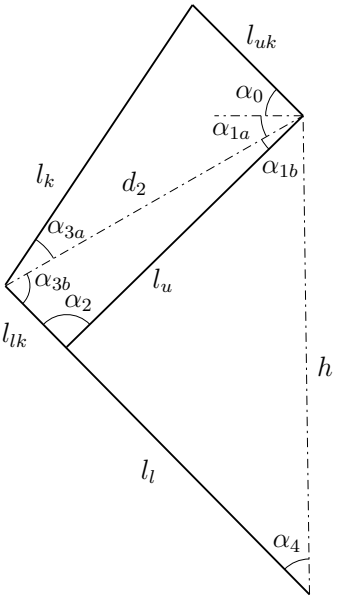


Figure 5.5: A drawing of the relevant quantities used in leg length initialization

5.2.2 | Connection to the Segway model

In order to use the controllers defined for the Segway model (such as our baseline LQR) on the MuJoCo model, we need a way to obtain the state variables of the equivalent linear model, as shown in Figure 5.6. On the real robot, the forward velocity $\dot{x}$ is proportional to the average of the angular velocities of the wheels, while the angular velocity $\dot{\psi}$ is is proportional to their difference.

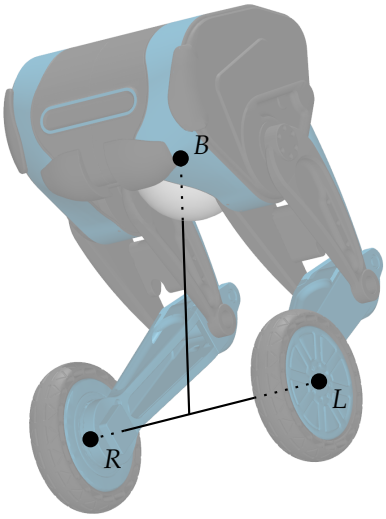


Figure 5.6: The segway we fit to the 3D model. The partially visible white ball is the computed COM of the robot.

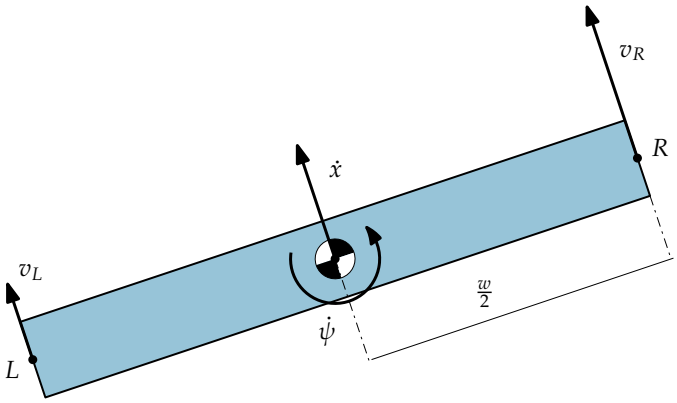


Figure 5.7: An equivalent rod we can use to properly define the meaning of $\dot{x}$ and $\dot{\psi}$.

In the end, I chose a different approach in the non-linear model for two reasons. First, due to the way the robot is assembled in MuJoCo, the wheel joint velocity does not only depend on the angular velocities of the wheels but also on the relative positions of the legs (wheels can be stationary with nonzero wheel joint velocity for example when the robot is falling forward), which makes this method more complicated. The second reason is more important however – such approach is inherently imprecise in case of any slippage.

Since we are working with a simulator, we can get what we need in an easier way. I mounted two artificial linear velocity sensors to each of the wheels. By projecting their output vectors onto the $xy-$plane (the $z-$component should be small anyways when driving in the plane), the situation can be modeled by Figure 5.7, where we know the velocities of the left and right wheels, v_L and v_R. Note that due to the structure of the robot, the two vectors will be colinear.

We would like to find a decomposition of the movement into $\dot{x}$ and $\dot{\psi}$ such that

$$v_L = \dot{x} - r\dot{\psi}, \tag{5.10}$$

$$v_R = \dot{x} + r\dot{\psi}. \tag{5.11}$$

It is easy to see that the solution to the above is

$$\dot{x} = \frac{1}{2}(v_L + v_R), \tag{5.12}$$

$$\dot{\psi} = \frac{1}{2r}(v_R - v_L). \tag{5.13}$$

Of course, the simulation runs in three dimensions and it is necessary to convert the incoming data into this description. To do this, we first need a way to find the forward direction f of the robot. We can do this by computing the cross product of the imaginary axle $a := R - L$ and the vector from center of rotation to the body, $r := B - (R + L)/2$,

defining

$$f := a \times r = (R - L) \times \left(B - \frac{R + L}{2} \right). \tag{5.14}$$

When we have f, we can compute the quantities of interest as

$$\dot{x} = \frac{1}{2} \operatorname{sign}((v_L + v_R) \cdot P_{xy} f) \|v_L + v_R\|, \tag{5.15}$$

$$\dot{\psi} = \frac{1}{2\|r\|^2} \operatorname{sign} \omega \cdot z \|\omega\|, \tag{5.16}$$

$$\text{where} \quad \omega = \frac{1}{\|r\|^2} (v_R - v_L) \times r, \tag{5.17}$$

$z = (0, 0, 1)$ and $P_{xy} = \operatorname{diag}(1, 1, 0)$ is the projection matrix onto the $xy-$plane.

Computing a pitch angle $\bar{\varphi}$ is easier, now that we have everything defined. For the equivalent model to make any sense, we assume that $\bar{\varphi} \in [-\pi/2, \pi/2]$, in which case it can be computed as

$$\bar{\varphi} = \arccos(f \cdot z) - \frac{\pi}{2}. \tag{5.18}$$

Additionally, there is an offset of approximately $\varphi_0 = -0.1$ rad (found by reading out the value of $\bar{\varphi}$ in a stabilized position in simulation), so we define the final pitch angle as $\varphi = \bar{\varphi} + \varphi_0$.

A good sanity check for the accuracy of the resulting model was to compare this constant with the real robot. Indeed, a very similar value can be found in the controller currently governing SK8O's movement, which is certainly a good sign.

The last state variable missing now is the pitch rate, $\dot{\varphi}$. To obtain its value, I mounted an artifical gyroscope at the COM of the robot in simulation and read the value of the appropriate axis.

5.3 Balancing

As with the linear model, we start with a simple(r) task – balancing. Besides the much higher number of degrees of freedom (DOFs), there is another complication: we would like to control the height (hip angle α_2) of the robot. In other words, even this task now has a reference h_{ref}.

5.3.1 Observation and action space

At this point, we are at an ideological crossroads, so to speak – what features will we use to train the networks? Should we use those identified as the state variables in Section 4.1, or just use the data from the sensors and let the agent make sense of it on its own? Well, we will test both. For now, let us define four different modes of operation which we will test.

For compatibility with the linear simulation, we will use the `segway` mode. Optionally, the environment can lock the hip positions in place to make it easier to test controllers which only assume control over the wheel motors. Alternatively, there are PD controllers

	dim	segway	segway+	sensors	all
Segway state $\bar{q}$	4	+	+	-	+
references $x_{\text{ref}}, \dot{\psi}_{\text{ref}}$	2	+	+	+	+
hip motor positions and velocities	4	-	+	+	+
wheel motor positions and velocities	4	-	-	+	+
IMU data	6	-	-	+	+
body orientation	4	-	-	+	+
roll θ	1	-	-	-	+
hip angle reference h_{ref}	1	-	+	+	+
hip action u_h	2	-	+	+	+
wheel action u_w	2	+	+	+	+

Table 5.2: The `sk8o_full` environment supports four different modes, each containing different observations and actions.

prepared for both 50 Hz and 1 kHz control frequencies which can take control over the hips if required.

As a very minimal extension, there is the `segway+` mode, which additionally includes the hip motor positions and velocities in the observation and allows for hip control. The `sensors` mode can be used to train the model with the data coming directly from the sensors and the exceedingly eloquently named `all` mode returns all the known data. A comparison of all the modes is shown in Table 5.2.

5.3.2 Initial conditions

During training, the starting length of both legs is sampled from a normal distribution $h_0 \sim \mathcal{N}(0.225, 0.02)$. The body can optionally can also be randomly rotated with the default values being set so that in most cases, the LQR controller can recover, though this is not used in practice. The $z-$position of the robot is then adjusted so that at least one of the wheels (in the case of rotations) is touching the ground at the start of the simulation.

This is in accordance with the start-up procedure with the real robot, which is also being held in this standard position when the controller is turned on. Even though in-the-air initializations might be possible with raw sensor data, I did not explore this avenue.

5.3.3 Rewards

The reward function will be largely the same as in Section 4.3.1 – in fact, the plan is to reuse the coefficients we have already found. We will add three additional terms due to the additional degrees of freedom of the system. The new function is now

$$\rho_{\text{balance,mjc}}(\bar{q}, r, u, \theta, \bar{h}) = \rho_{\text{balance}}(\bar{q}, r, u) - c_h \|u_h\|^p - c_\theta |\theta|^2 - c_\alpha \left\|\bar{h} - h_{\text{ref}}\right\|^p, \tag{5.19}$$

where θ is the roll angle, $\bar{h}$ is the average leg height (why we use the average will become clear in velocity reference tracking) and u_h is the vector of hip actions (left and right). Note that even though the penalization term contains $\bar{h}$ (which is what we ultimately

care about), internally, the agent receives the reference for the hip angles, because we receive hip motor positions and velocities from both the model and the real robot.

To find the new coefficients, we will take a different route than in the two previous cases. Based on a few experiments, I set $c_\theta = 0.05$ and $c_h = 0.01$ in training (and $c_\theta = 1$ and $c_h = 0.1$ in evaluation so that they have an effect there). A justification for these low values could be that their role is only to nudge the training in the right direction and we will satisfy ourselves knowing that the models learn to stabilize the robot properly.

I admit that this approach is significantly less systematic than before. However, training in MuJoCo takes considerably more time – some of the velocity reference tracking models took three days to learn.[3] For this very reason I only tested each run with three seeds instead of five in the following figures.

With those two values set, we try to find a good value for c_α, as always with SAC with default hyperparameters and no noise. Out of $c_\alpha \in \{5, 10, 20, 50\}$, I chose $c_\alpha = 20$, based on the experiment included in the appendix (Figure F.6). Unfortunately, due to high variance, it can be hard to convey the information you can get using an interactive visualization, so to put my reasoning into words: this coefficient was not the fastest to learn but was more stable in the evaluation afterwards.

5.3.4 | Robustness against inconsistent rewards

In [41], the authors show that episode truncations can cause issues due to inconsistencies in reward estimations if not handled correctly. In response, the gym API has been changed to differentiate between reaching a terminal state (either falling or reaching target velocity in our case) and episode truncation (end due to time limit being reached).

When adjusting the code of the `sk8o_full` environment to accommodate these changes, I accidently flipped the order of these two variables in the return function. Apart from the (significant) time I spent looking for this bug, I also discovered an interesting fact about the DroQ algorithm.

Before we can get into that, however, we need to understand where lies the problem with episode truncations. The issue stems from the fact that during training of SAC (and others), the return R_T from Equation (3.6) is generally approximated as

$$R_T = \sum_{t=T}^{\infty} \gamma^t r_t = r_T + \gamma Q_{\tilde{\theta}_i}(\pi(s_T), s_T). \tag{5.20}$$

and when s_T is a terminal state, it is simply

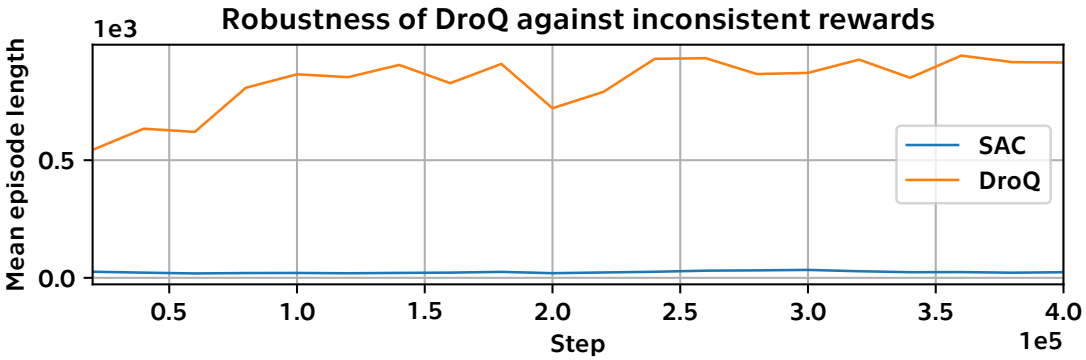
$$R_T = r_T, \tag{5.21}$$

because there were no more rewards in the episode.

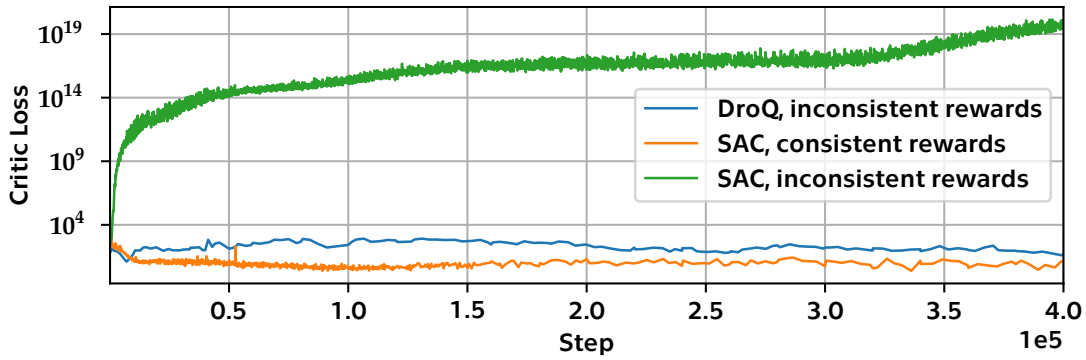
When s_T is reported to be terminal only sometimes (as it happens when we do not handle truncations properly), the data is inconsistent and the accuracy of the $Q-$function is decreased. Though this difference looks very subtle, it can be noticeable in practice, as has been observed by [41] and accidentally by me.

[3] All training was performed on an Intel Xeon 4410 CPU @ 2.1 GHz with two cores allocated.

Interestingly, while vanilla SAC failed to learn anything with this bug in place, DroQ was much more robust against this error, as shown in Figure 5.8. If we look at the average loss of the $Q-$functions in Figure 5.8b, we can see that for DroQ, though it is considerable and does not go to zero, it does not grow without bounds like in the case of SAC, possibly due to the inconsistencies in rewards.



(a) SAC failed to learn anything but to fall immediately.



(b) While the critic loss of DroQ was still an order of magnitude larger than standard, it was bounded.

Figure 5.8: DroQ learned to balance SK8O reasonably well even with inconsistent rewards, unlike SAC.

5.3.5 | Going deeper

With the bug out of the way, I started training the algorithm with variable height reference. I soon found out that unlike in the linear environment, the Deep Dense Architectures in Reinforcement Learning (D2RL) network structure is a fair contender for the standard MLP. As Figure 5.9 shows, D2RL is a significant improvement over vanilla SAC in this case.

Why is that? If we look at the structure of the networks, one might wonder whether the cause for that could be that the skip connections allow the deeper layers to act upon the height reference earlier. Therefore, I trained a more shallow version with the same structure to test this hypothesis. The results of the experiment can be seen in Figure 5.10. Clearly, the advantage is not just due to the skip connections, the depth also plays a role.

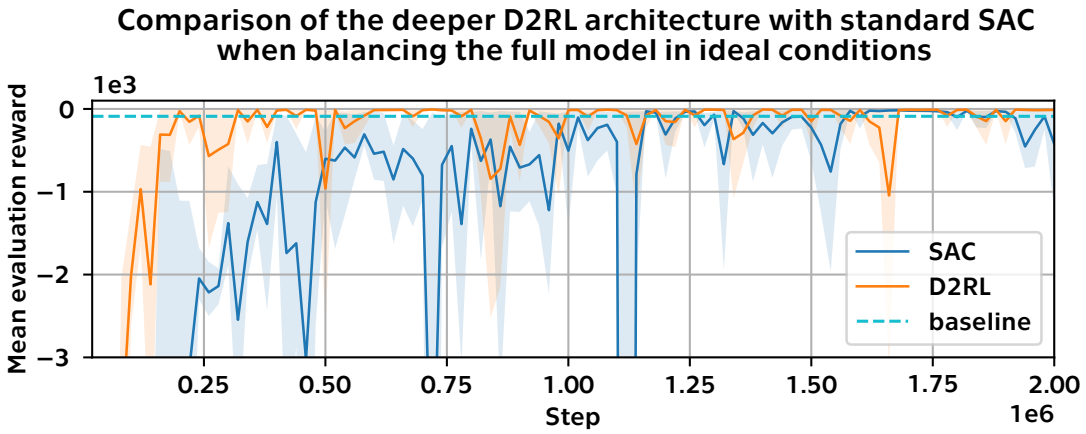


Figure 5.9: The D2RL network architecture learns faster and is more stable than SAC.

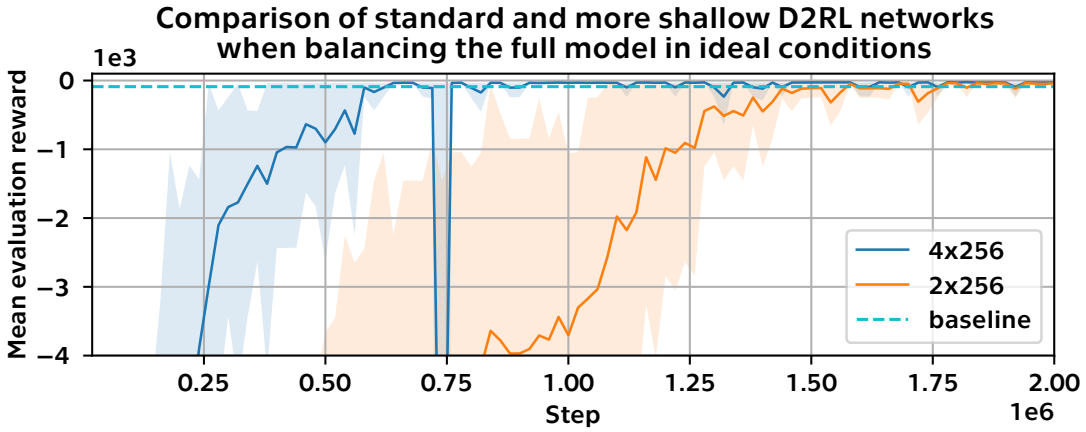


Figure 5.10: Depth is key in the improved performance of D2RL.

5.3.6 A case for feature engineering?

Having found the rewards and identified the benefits of D2RL, I then decided to check what observations are best for training the robot. I tested three out of the four modes presented in Table 5.2, leaving out the compatibility `segway` mode.

In my initial tests, the best performing one by far was the `segway+` mode. The reason for the question mark in the title of the section is another blunder of mine. The initial experiment was launched before the environment was finalized and due to another bug in the reported observations, which I later fixed, the results were significantly different. However, encouraged by the idea that the simplest mode performed the best, I did not think twice to recheck it before it was too late.

For integrity, I include the new experiment in Figure 5.11. Clearly, the `segway+` observation mode is the one that allows for the fastest learning. However, eventually, the mode containing also the raw data outperformes it (and the `sensor` mode is the slowest to learn, though it may potentially get there after the training ended). For this reason, I only consider the `segway+` more in future text with the implicit asterisk that it may not be optimal.

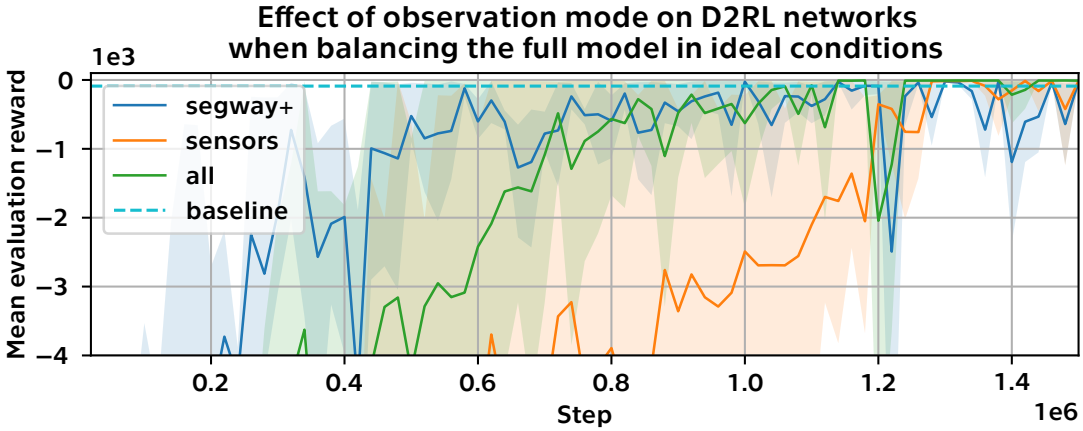


Figure 5.11: The `segway+` is the fastest to learn to balance but is eventually surpassed.

5.3.7 Adding noise into the mix

We conclude this section by experimenting with noisy measurements, as we have done before. The results are largely the same as in the case when no noise was present – D2RL is a significant improvement over vanilla SAC. The results of the experiment with D2RL, along with the best performing SAC and shallow D2RL can be found in Figure 5.12. The complete results of SAC are included in Figure F.7.

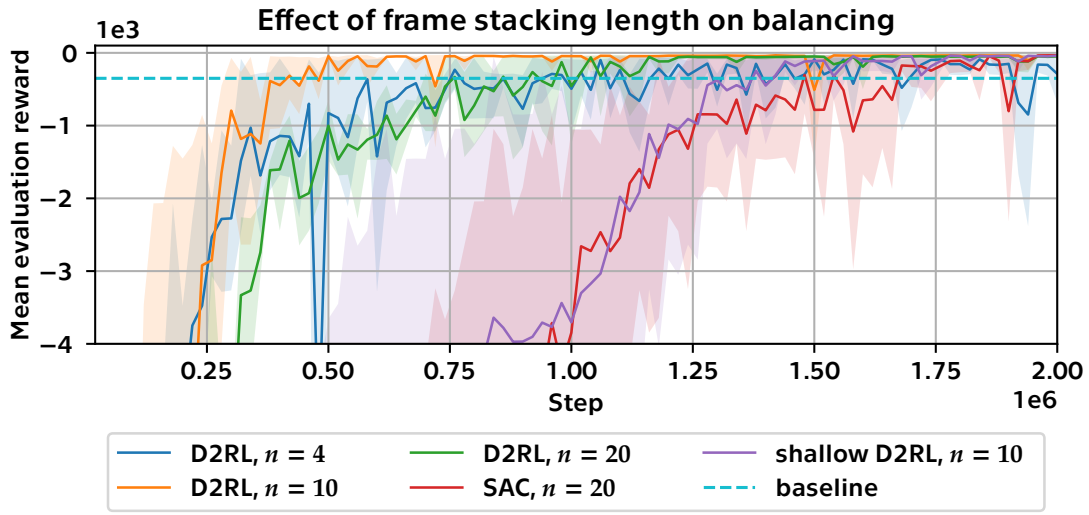


Figure 5.12: The advantage of using D2RL is even more pronounced when we consider noise.

5.4 Velocity Tracking

Equipped with experience we gained in the previous three similar sections, we will move faster here. For rewards, we will use the same coefficients as in Segway velocity reference, combined with the MuJoCo-specific coefficients we identified above. The only exception will be the roll coefficient, which we will lower to $c_\theta = 0.01$ in training and $c_\theta = 0$ in evaluation. The reason for this is that we would like to allow the robot to lean into the curves, which is beneficial especially at higher.

I argue that unlike c_ε, c_α can be left without change. Recall we had to change c_ε because the initial errors were too large with nonzero reference. This is not the case with height – the initial error distribution is the same. For the very same reason, we can leave the hip inputs without a change too.

I will only include the `segway+` mode here due to reasons mentioned in the previous section. However, I believe that using the `all` mode could allow for better performance and it might be a good direction of future work.

Based on the behavior of the baseline controller, I also increased the window of acceptable velocities in the *success* function from Equation (4.21) to 0.2 – double the original value.

5.4.1 Noise saves the day

Like in the case of Segway, including noise in velocity tracking improves the performance. Before I found that out, however, I started as usual with ideal conditions. One observation we can make about Figure 5.13 is that it would seem that the dominance of D2RL in the previous section was a random fluke, as its results are comparable with SAC in this setup.

Worried when I saw the problems when learning without noise, I commenced the part with noise by testing out a slightly simpler scenario – what if we only consider going forward and backward? That is, always setting $\dot{\psi}_{\text{ref}} = 0$. The results are depicted in Figure 5.14 and are certainly much more promising.

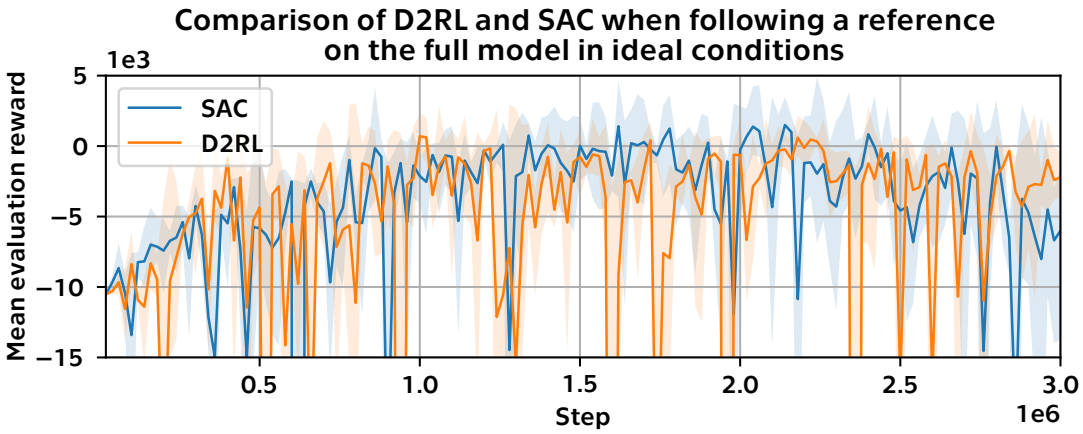


Figure 5.13: Notice that the mean evaluation rewards are around zero, so in about half of the episodes, the agents did not reach the target velocity.

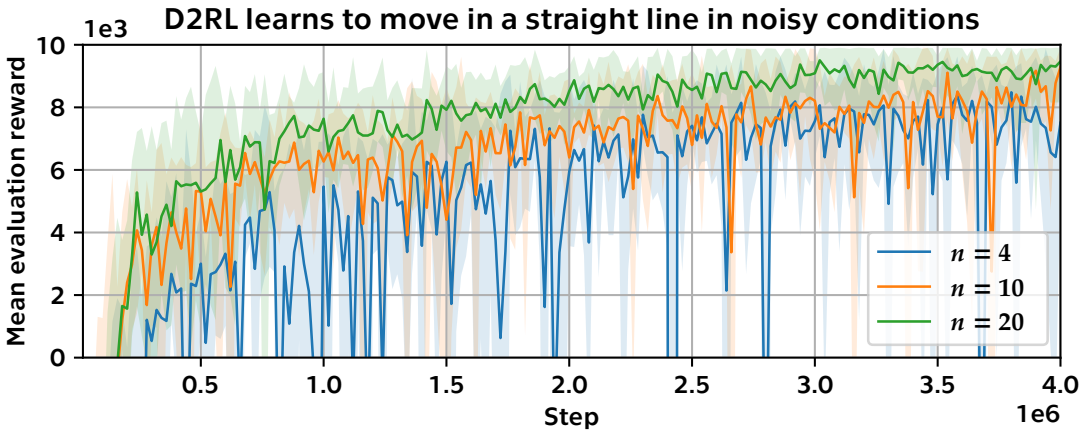


Figure 5.14: Learning to go in a straight line at a specified speed is a much easier task than tracking any reference. The results of SAC were nearly identical and are therefore omited.

If you were rooting for the relatively unknown D2RL, worry not, for when we consider the complete problem, it is again in the lead, as depicted in Figure 5.15. However, the scores are noticeably lower. It would seem that it is the combination of forward velocity and yaw rate that is the main problem.

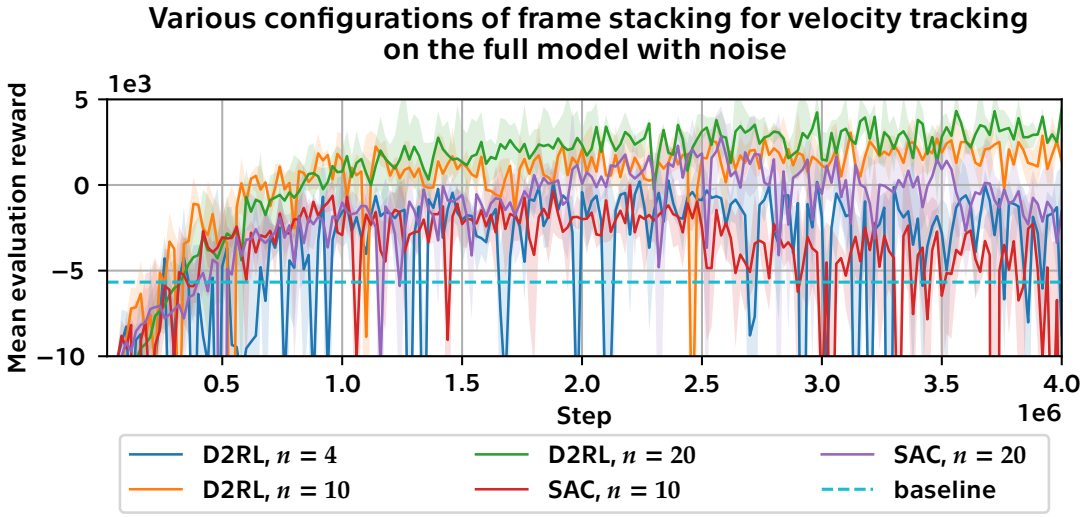


Figure 5.15: When learning to track velocities in a noisy environment, both SAC and D2RL are improved, with the latter coming out ahead.

5.4.2 | Learning with a teacher

As humans, we often learn by mimicking others. Although we have seen the algorithms learn to balance the robot and even reach a specified velocity, it might be interesting to see whether we can use a profficient controller to "guide" the agent towards good actions and speed up the learning.

For the first experiment, I tested the approach I outlined in Section 3.3.3: with exponentially decreasing probability $p = \exp^{-\lambda n}$, where n is the number of environment interactions, the action will not be selected by the agent, but rather by the teacher – the LQR. As we did not have a functional controller for this control frequency, I had to design one. The resulting controller managed to reach a more relaxed goal in approximately 30% of the episodes, which is a possible explanation for why learning in Figure 5.16 was not accelerated by the presence of a teacher (though it was decelerated when the teacher was "too active"). Still I am a little surprised that the reasonable policy did not improve performance even during the early stages of training.

Second, I tested the DroQ algorithm (discussed in Section 3.2.2), which boasts a much higher sample efficiency, in a slightly different scenario. Here, there was a number of trajectories pregenerated and the training started with a partially filled replay buffer. However, the results were so abysmal that I initially assumed that my implementation as incorrect. However, the algorithm performed as expected on the standard Humanoid environment, so it is possible that this learning strategy is simply not good due to the reasons outlined in Section 3.3.3.

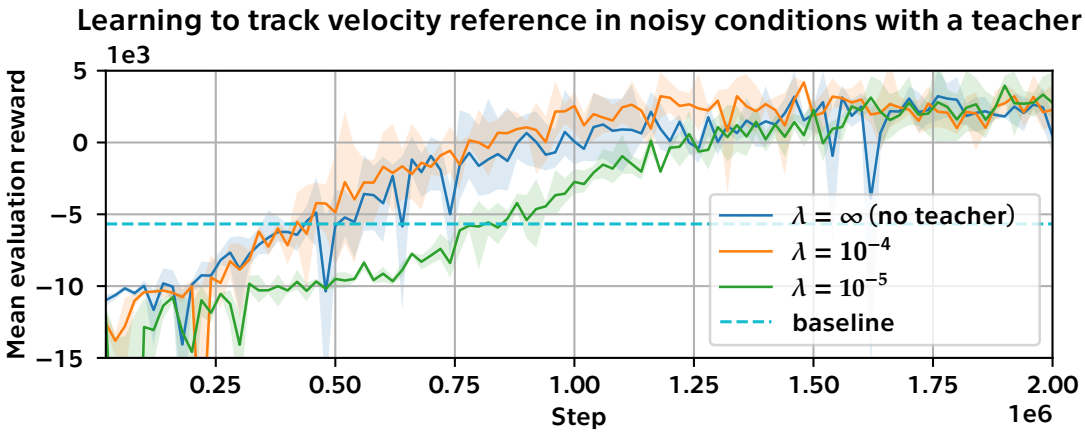


Figure 5.16: The presence of a teacher did not accelerate D2RL's learning. Note that in this case, the reference error window of Equation (4.21) was further increased to 0.4.

5.5 Verification

The first model we shall inspect was trained to balance SK8O in linear simulation, with noise and model randomization. In Figure 5.17, we can see a major problem with its behaviour – it is extremely sensitive to the offset pitch angle, denoted by φ_0 in Section 5.2.2. We can see that depending on the value of this angle, the robot is leaning (and therefore moving) either forward or backward. This is problematic because φ_0 changes in practice with changing height but also in case of any changes in the robot, such as when it uses a heavier battery.

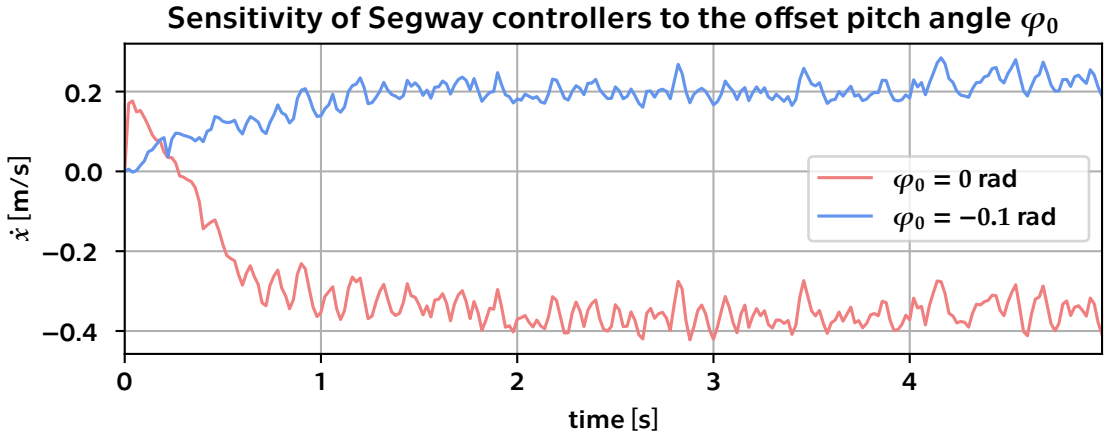


Figure 5.17: Depending on the offset pitch angle φ_0, the balancing Segway controller moves either forward of backward.

As they say, "hindsight is 20/20". Nonetheless, I should have predicted that this problem could arise – after all, one of the few invariants of the simulation with parameter randomization in place was the fact that the system is stable when $\varphi = 0$. In an effort to mitigate this problem, I devised a training scheme, in which φ_0 is randomized as $\varphi_0 \sim \mathcal{N}(0, 0.1)$. However, while this fixed the problem in the sense that changing φ_0 does not alter the behavior, the controller performs poorly either way, as seen in Figure 5.18. Training with zero penalty on the pitch angle φ did not help with the issue either.

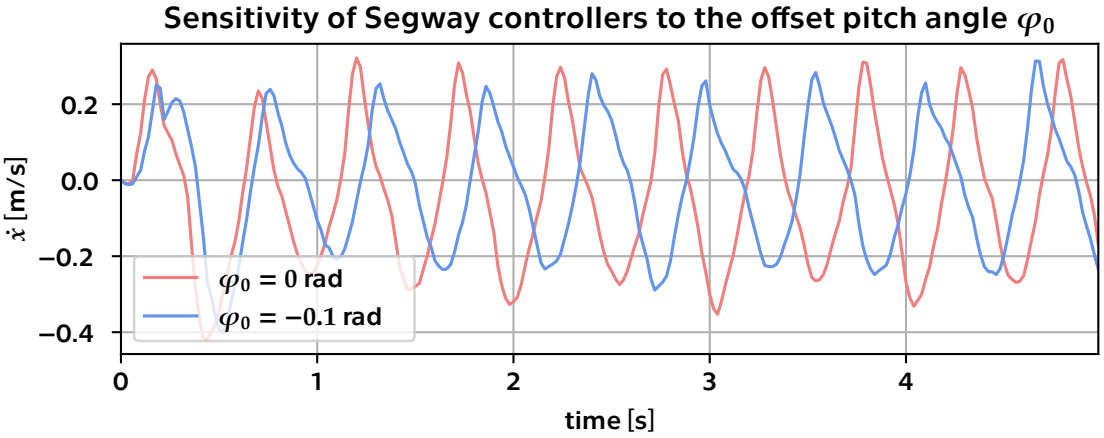


Figure 5.18: Using variable pitch offset φ_0 in training causes oscillatory behavior.

Out of the linear simulation-based controllers, the one that performs the best was trained for velocity tracking, shown in Figure 5.17. This is not very surprising, given that it outperformed the balancing agent even on the Segway simulation. However, it still suffers from the same issue – sensitivity to φ_0, which I was unable to suppress.

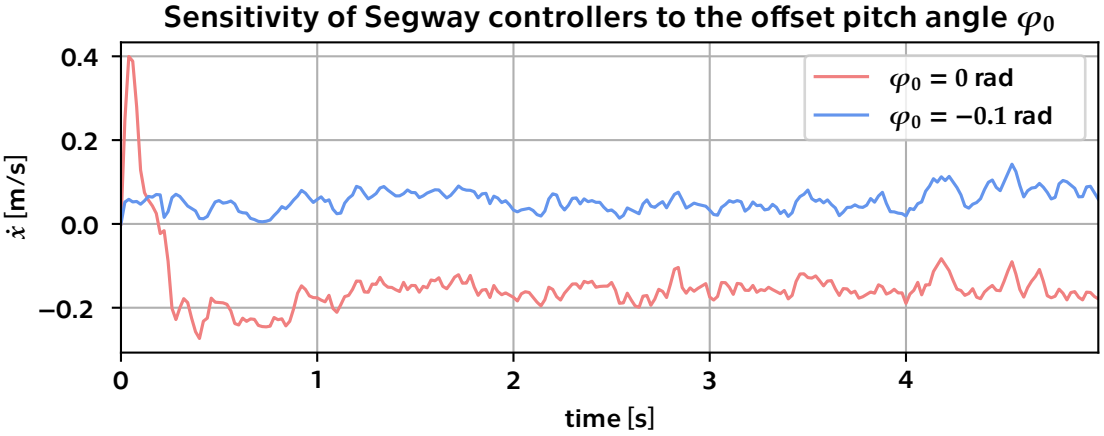


Figure 5.19: The velocity reference controller show the best performance out of those trained on the linear system - its sensitivity to pitch angle offset φ_0 is the lowest.

The reason why we decided to train RL agents on a full 3D model of robot was to allow it to control the hips. Therefore, the balancing results in Figure 5.20 include variable height reference. While the agent tracks the requested mean hip angle accurately, it looks as though it tends to lean to one side. This was likely caused by the roll coefficient being too low in the reward function.

5.5.1 | Velocity tracking

The agents trained in the full simulation do not exhibit the same level of "mastery" as those trained on the Segway model as illustrated by Figure 5.21. However, at least anecdotally, the RL agents did learn to lean into the curves, as shown in Figure 5.22, which goes to show that they do have at least some understanding of the system and it is, in my mind, at least a partial success.

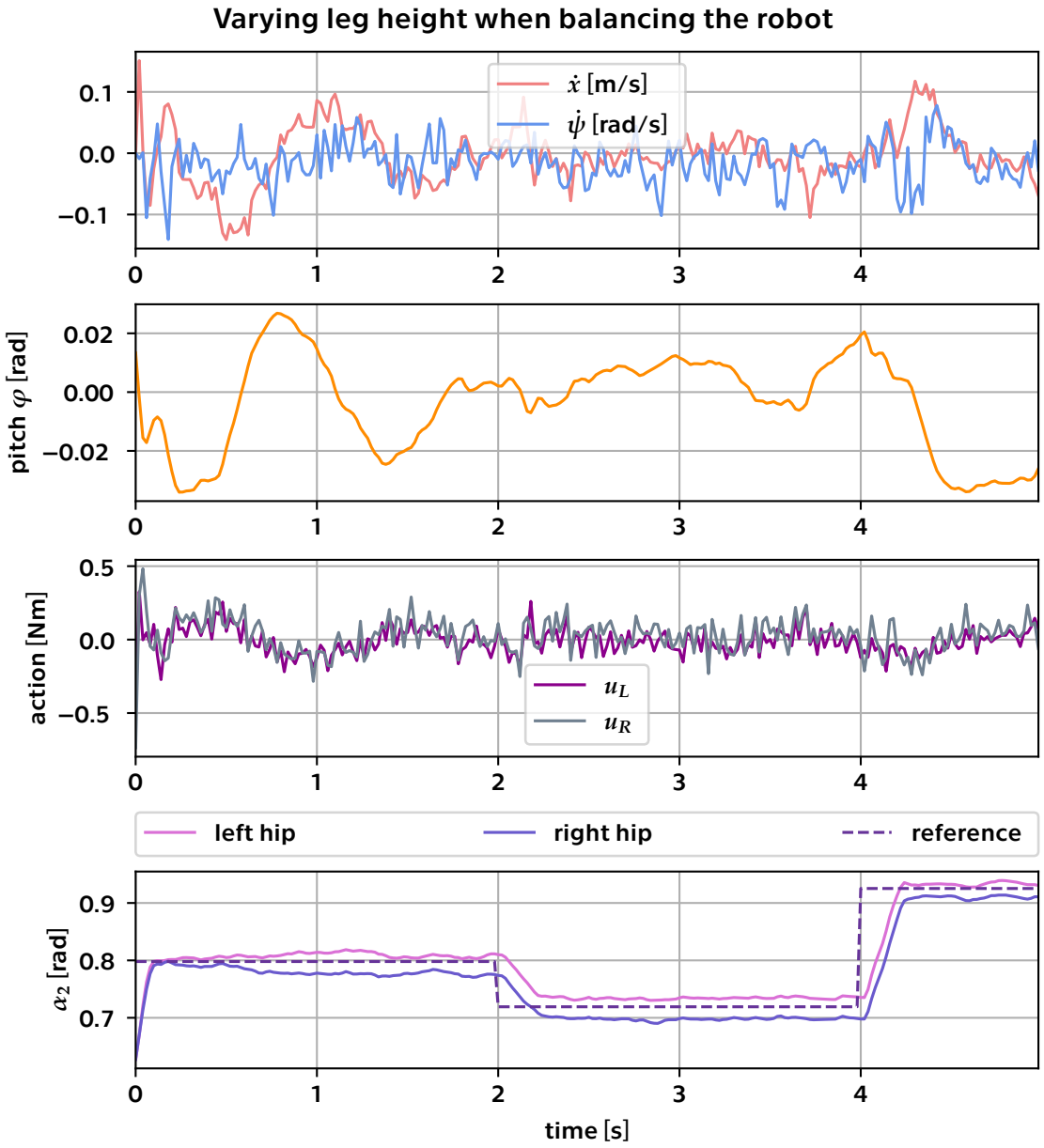


Figure 5.20: A full view of balancing with noisy model and perturbations by an RL controller trained in MuJoCo.

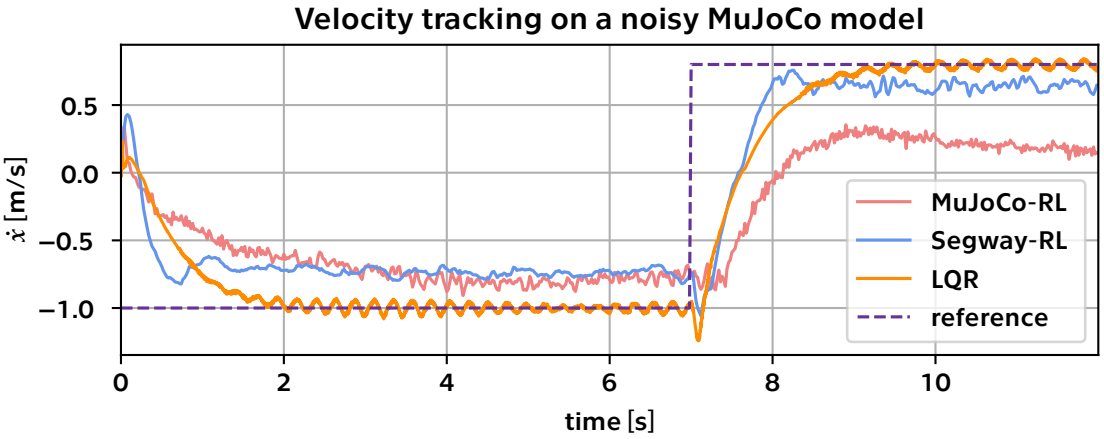


Figure 5.21: The policies trained in MuJoCo are "beat at their own game" even by the controller trained on the linearized Segway model.

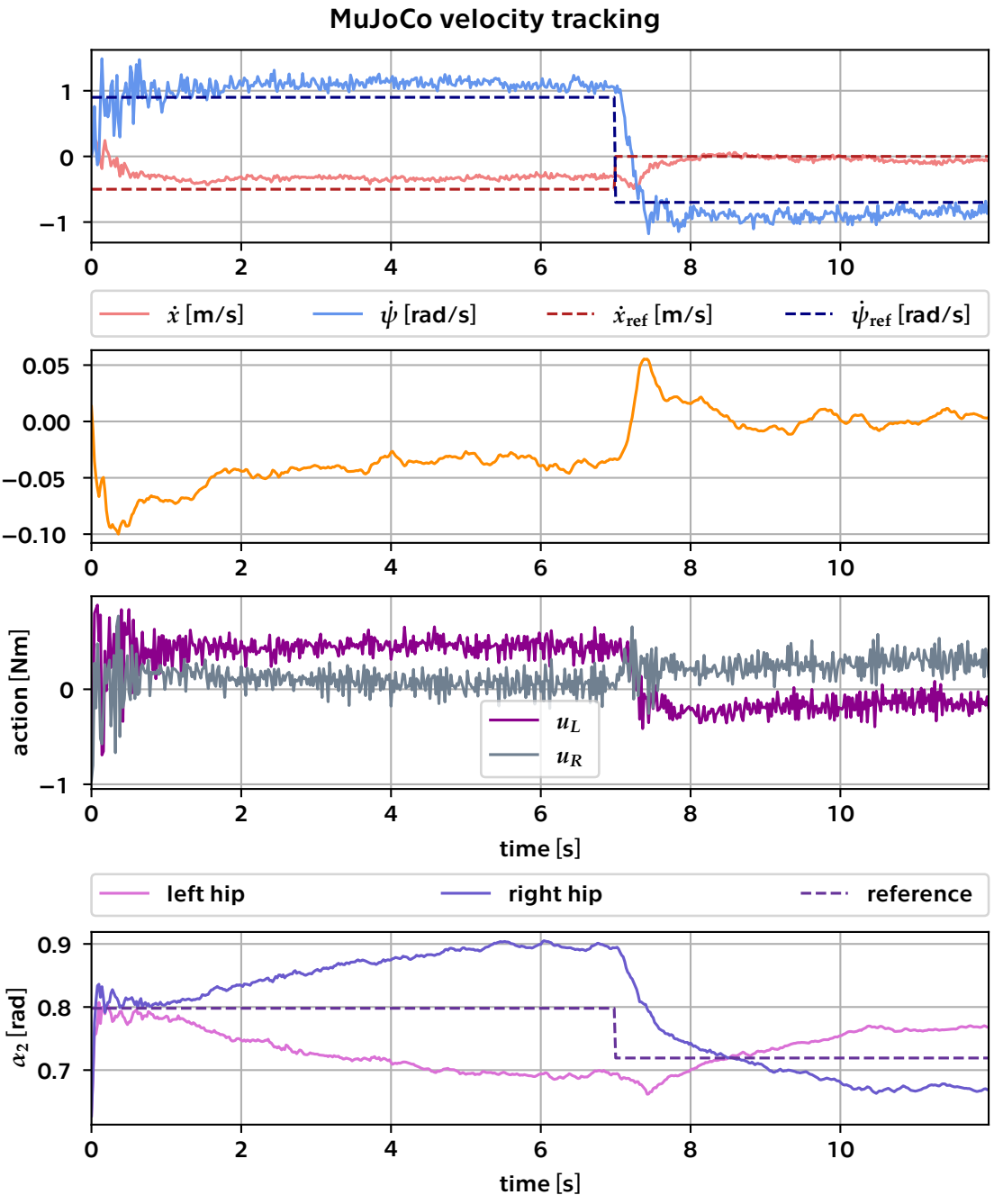


Figure 5.22: A full view of velocity tracking with noisy model and perturbations by an RL controller trained in MuJoCo. Notice that the robot "leans into" the turns.

Chapter 6

Deployment onto SK8O

Making the leap from simulation to the real world is a complex process, even with classical methods. This is doubly true for Reinforcement Learning – neural networks tend to overfit to the training environemnt (as illustrated by [19]) and they are significantly more demanding, both from software and hardware point of view, than essentially all of their standard counterparts.

Running the models

So far, SK8O has been mostly controlled from a Teensy board (recall Figure 2.2), although Petr Brož has made progress in the past year to allow for control from the much more capable ODroid board, which runs Ubuntu Linux. As of May 2023, there are still some issues to be ironed out that may manifest themselves in our experiments but thanks to his work, we were able to deploy the models to ODroid and not worry too much about the extremely resource-constrained Teensy.

The control on both the Teensy and Odroid boards was implemented in C++, though an interface to Python for high-level features, such as a web dashboard and Xbox control, had been implemented. As discussed previously, the models we trained in Chapter 4 and Chapter 5 were exported to the ONNX format with the hopes that that would allow an easy transition to the real robot, with a control loop in C++.

Unfortunately, we had underestimated the difficulty to use these complex libraries on the ARM architecture. Binaries for the x64 architecture are readily available from official sources for both Torchscript (a method for creating C++ programs from a PyTorch model) and the ONNX Runtime, which is not the case for ARM. After many hours of trying to compile the libraries from source, we[1] decided that C++ was not the way to go here.

In the end, the control loop runs in Python.[2] Of course, this has some drawbacks – Python is known for many things but being fast is certainly not one of them. Consequently, the control loop cannot reliably run faster than at 50 Hz. Therefore, running the NNs was not even the bottleneck (the default SAC agent runs at >10 kHz).

[1]The deployment was greatly accelerated by the help of Martin Gurtner who had implemented a large part of the current SK8O codebase.

[2]An interesting sidenote is that our unsuccessfull attempts at compiling either torch or the onnxruntime took longer than implementing the control in Python.

Discrepancies between the model and the real robot

Any simulation model is an idealized version of the real one and those presented in the preceding chapters are no different. Let us now take a moment to go over the discrepancies that proved to be most influential to the deployment.

The first minor hiccup is the fact that the state vector from Section 4.1 is not the one used in the robot. Besides a reordering of the states, states $\dot{x}$ and $\dot{\psi}$ are replaced by the sum and difference of wheel angular velocities ω_L, ω_R. Therefore, it is necessary to compute the state used for predictions as

$$\dot{x} = -\frac{r}{2}(\omega_L + \omega_R) \quad \text{and} \quad \dot{\psi} = -\frac{r}{d}(\omega_R - \omega_L), \tag{6.1}$$

where r and d are wheel diameter and axle length, with values found in Appendix C. Note the negative signs due to different wheel orientation. For reference, the full state transformation can be described by

$$\bar{q}_{\text{robot}} = \begin{bmatrix} 0 & 0 & -d/r & 0 \\ -2/r & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bar{q} \quad \text{and} \quad r_{\text{robot}} = \begin{bmatrix} 0 & -d/r \\ -2/r & 0 \end{bmatrix} r. \tag{6.2}$$

Additionally, the angular velocities are quantized to steps of approximately $6/91$ rad s^{-1}.

Likewise, it is necessary to convert the data coming from the hip motors. They are again oriented in the opposite direction and geared up by a factor of $\gamma = 16.5$. Lastly, during the homing procedure before the controller is started, motor positions with fully extended legs are recorded. In my experiments, this value was found to be equal to $\alpha_{2,0} = 64°$. The complete transformation is then given by

$$\alpha_2 = \alpha_{2,0} - \frac{p}{\gamma}, \tag{6.3}$$

where p is the positional output of the respective motor. Apart from the offset, an analogous equation can be used to obtain motor velocities.

6.1 Experiments

For the third time in this text, we will get to see sample trajectories from the models. In this case, we will make one change: we do not differentiate between balancing and velocity reference tracking. The reason for this is simple – the best performing balancing controller is in fact a velocity reference controller, as seen in Figure 6.1, where it significantly outperforms the best balancing controller I found.

My theory for this is that the controller that trains for velocity tracking explores the state space better due to its task, while the balancing one only truly gains experiences in the neighborhood of the origin. Due to the inevitable differences in dynamics, the balancing controller thus finds itself in situations it had not seen before and has problems stabilizing the system. Another explanation might be that the balancing controller is

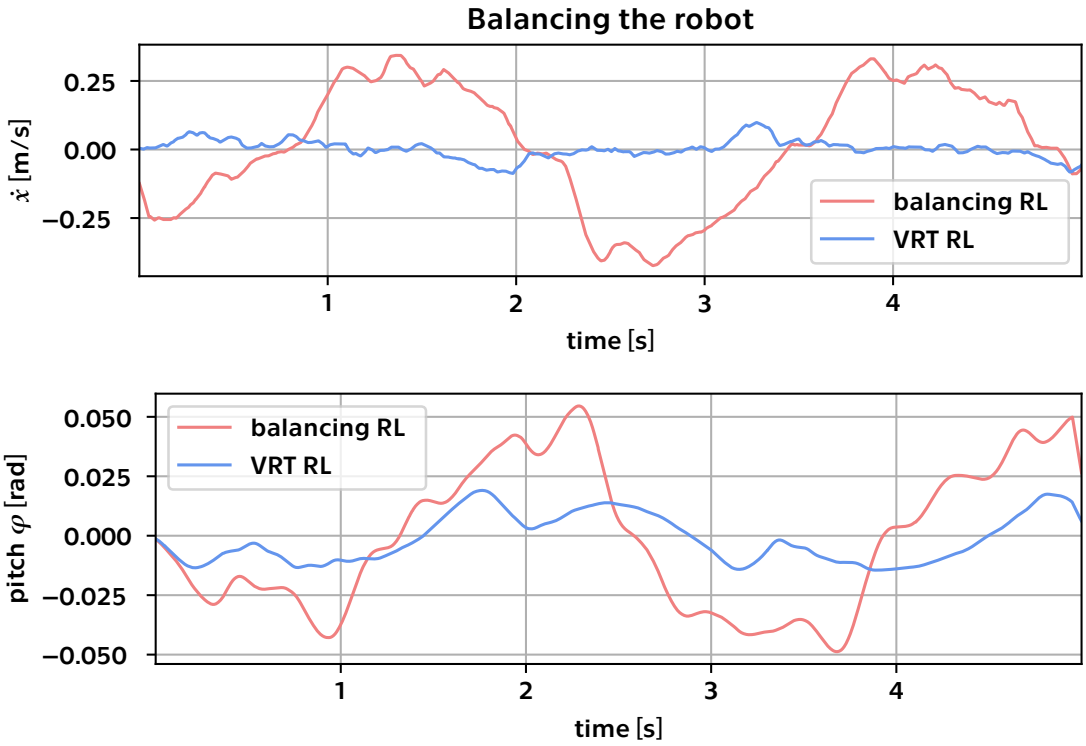


Figure 6.1: The velocity tracking controller is superior to the one trained only for balancing.

simply overfit, which the velocity reference either is not or is less so thanks to more diverse training dataset. These are, however, only theories and would require more time to be explored properly.

I was pleasantly surprised by the performance of the Segway-based controllers, as I had not expected them to work without any fine-tuning. While they still exhibit the sensitivity to pitch angle offset φ_0 we discovered in simulations, I would argue that the issue is manageable in practice.

The controller is even able to survive being pushed, as illustrated by Figure 6.2. Interestingly, it is much more resilient towards pushes that would make it fall backward. This could be connected to the sensitivity to pitch offset variation – it is entirely possible that the one used in the robot leads to stable position having nonzero pitch angle φ, biasing the controller to lean forward.

A minor issue we ran into was a very jittery movement, accompanied by strange noises coming from the motors. Though it is also present in the LQR controllers when being run on the ODroid, the RL agents exacerbated the issue. In an effort to try to solve the problem, I retrained a new set of Segway-based agents with the quantization built-in and, subjectively, this improved the behavior.

There is one more interesting experiment we can look at, which has to do with input shaping (or the lack thereof). When requiring a linear controller to suddenly move at a drastically different velocity, it is best to shape the reference change for example with a low pass filter, because the response is roughly proportional to the reference error. Theoretically, the RL agents should be immune to such issues – in the end, the were trained without input shaping.

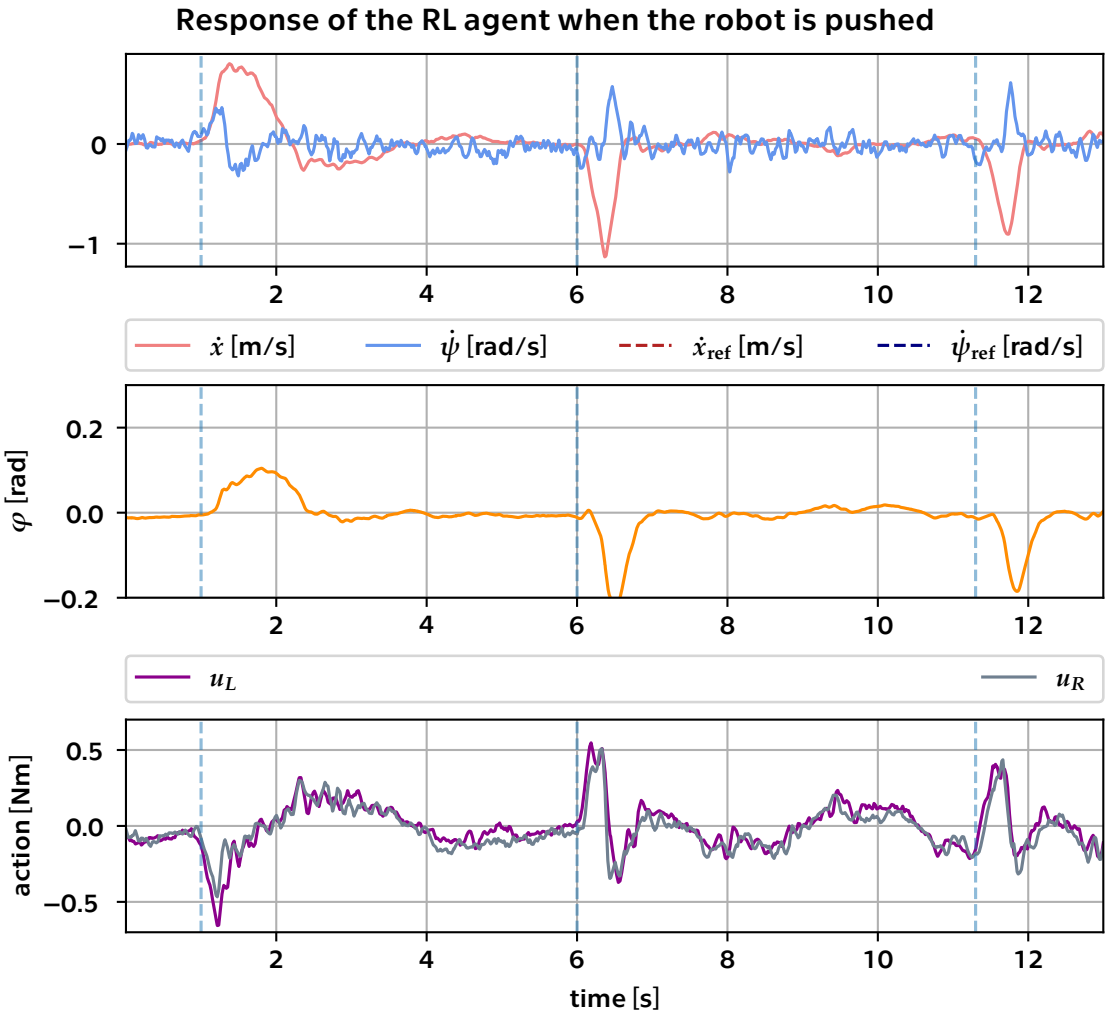


Figure 6.2: Pushing the robot back and forward does not make it fall.

The results are presented in Figure 6.3. The RL controller manages to stabilize the robot with a much larger jump in reference. This is remarkable also due to the fact that the agent has not experienced such references during training at all (recall that the maximum velocity was 1 m s) and has likely never explored this part of the state space.

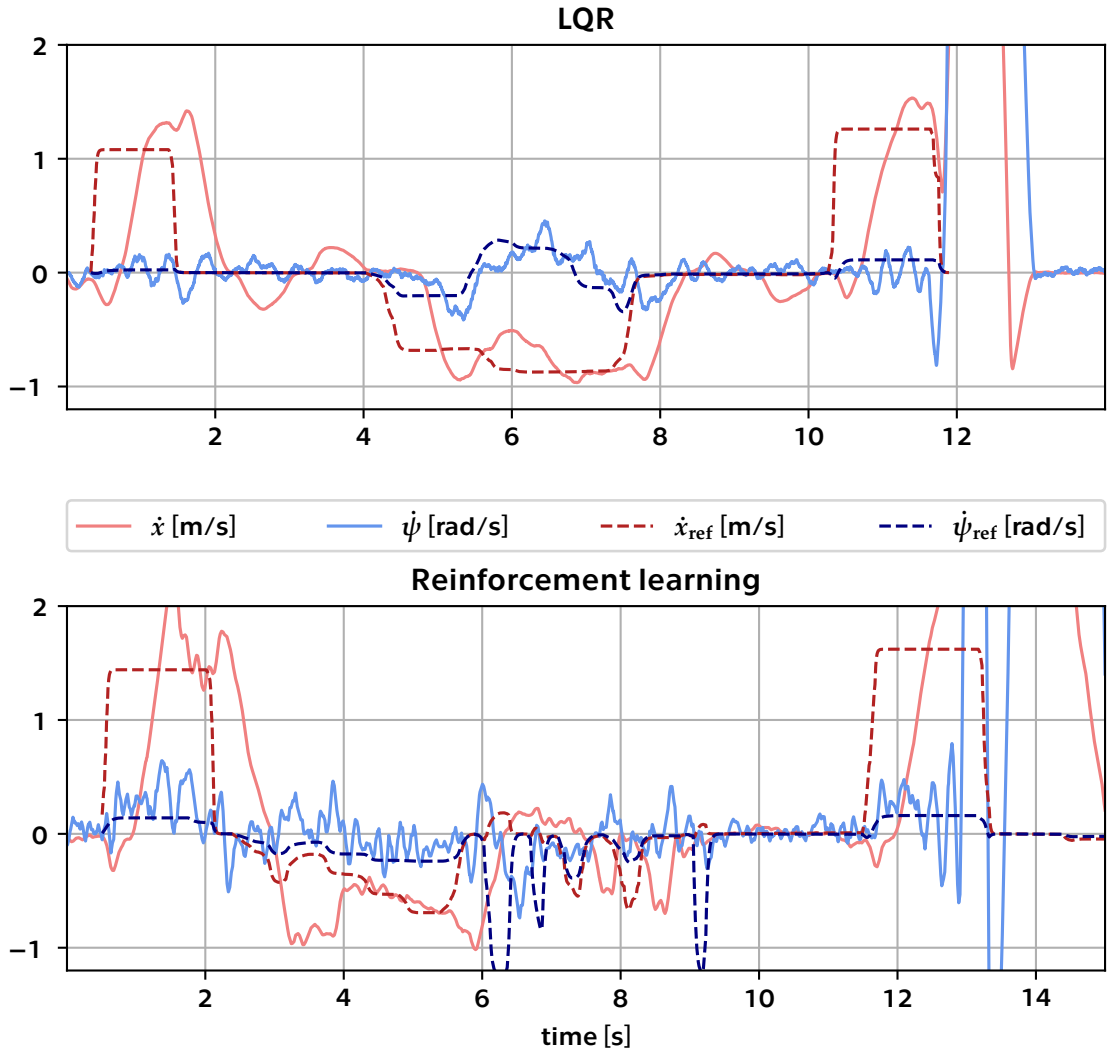


Figure 6.3: The RL controller can withstand larger jumps in reference than the LQR used on SK8O. The important areas in the figure are the responses to reference steps at the beginning of each timeseries and at around 10s in the case of LQR and 12s for RL, both of which are followed by a fall. Notice that the RL agent is able to respond to a change greater than the one that caused the LQR to fall.

Finally, in Figures 6.4 and 6.5, you can compare the behavior of the RL Segway-based agent to the LQR from Equation (4.9) running at 1 kHz. The RL policy has a nonzero steady-state error and is more sensitive to noise in general. On the other hand, it is subjectively less sensitive to changes in height of the robot. I think it is fair to admit that the LQR controller is still superior in most cases. However, just the fact that we can compare the RL-based controller to one running at 20x the frequency is, in mind, an astounding achievement.

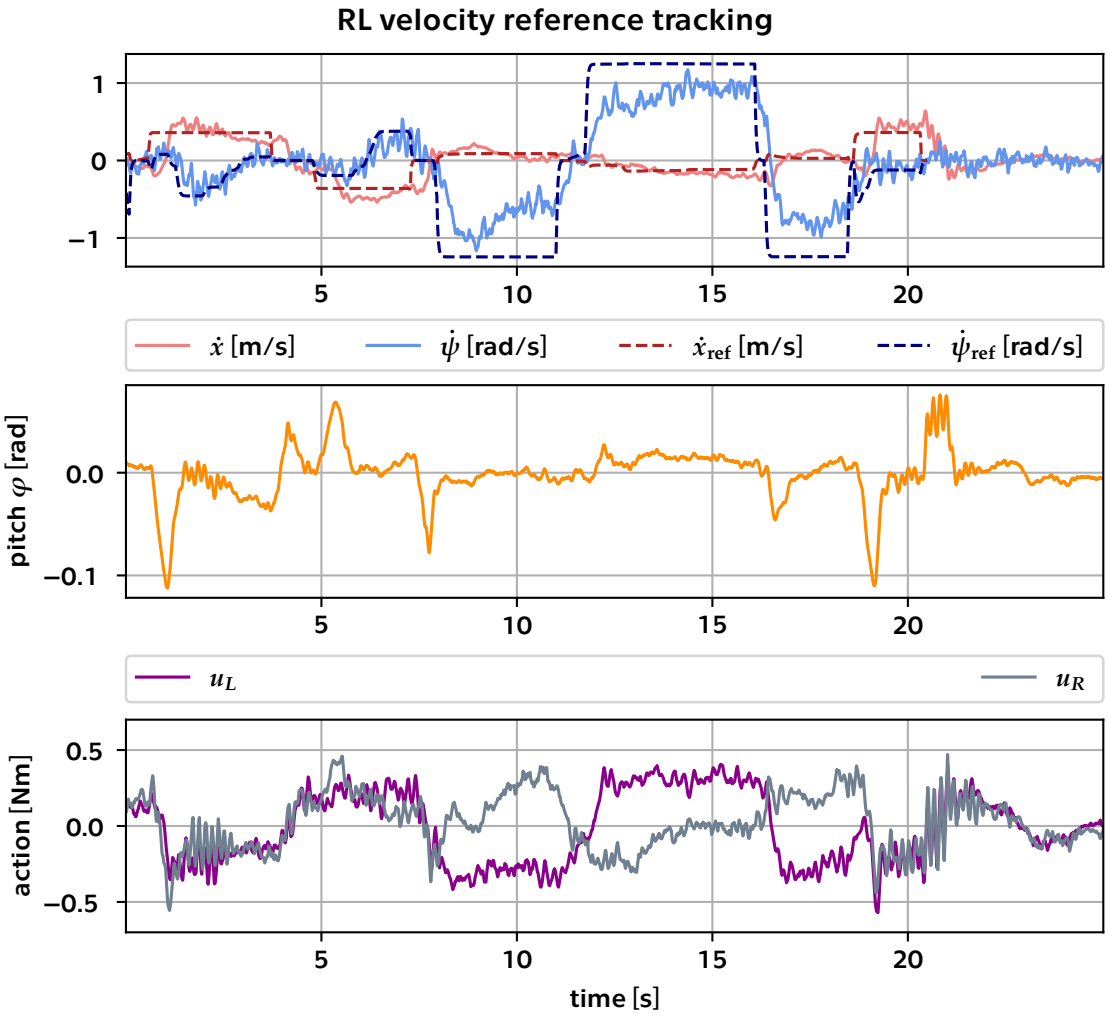


Figure 6.4: Measurements from the real robot when controlled by SAC agent trained on the linearized Segway model.

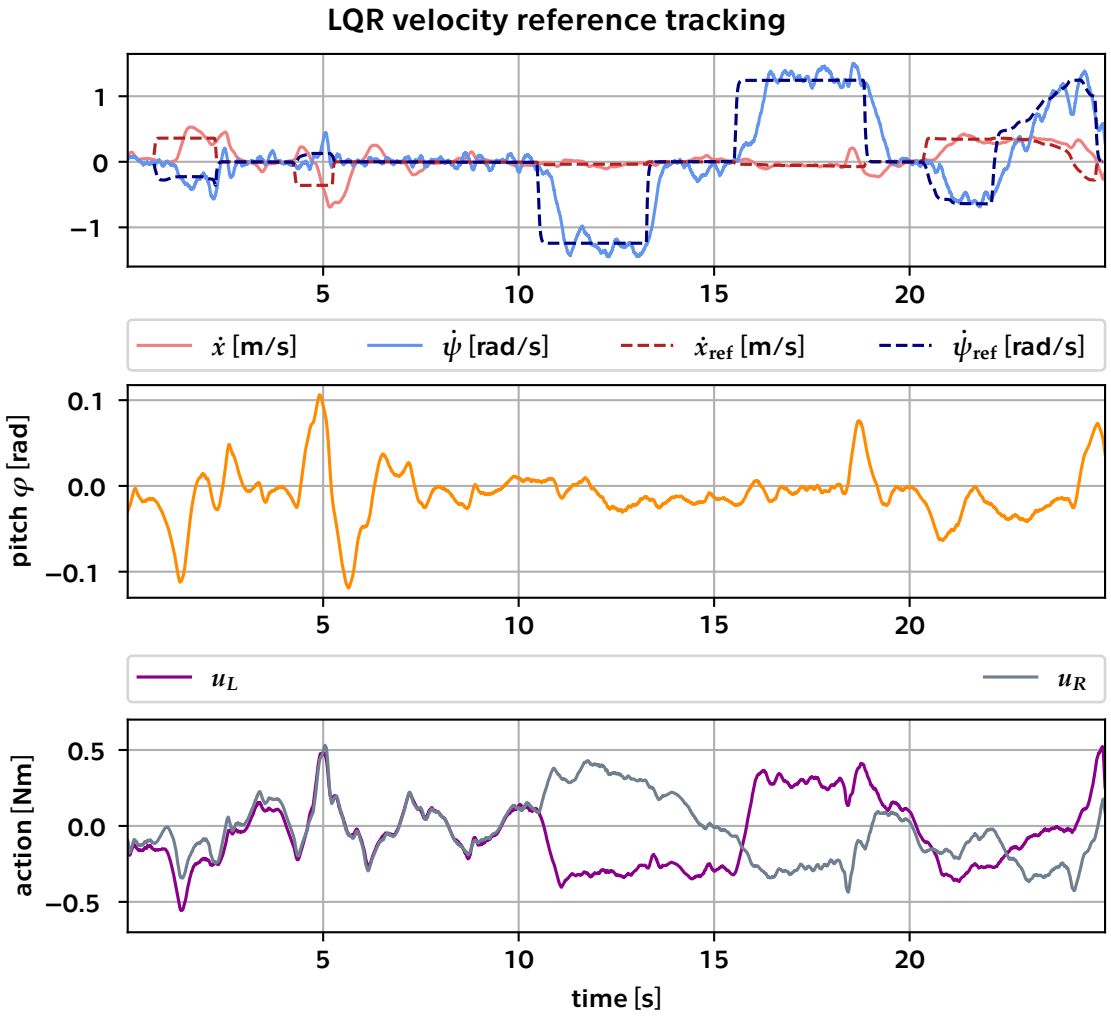


Figure 6.5: Measurements from the real robot when controlled by the LQR currently used.

6.1.1 MuJoCo-based controllers

Unfortunately, the MuJoCo controllers failed to deliver. To an extent, I had expected this – we have had issues with LQR controllers working in simulation and not on the real robot and vice versa, so we suspected that the simulation was not entirely accurate. Because the controller from [2], which was also the source of almost all physical paramers, works very well in simulation, I believe the issue might lie with the system identification. Since then, the robot has undergone severl modifications: the RealSense camera was mounted, the wheels have been swapped and some of the parts are worn out (especially the knee springs), so it does not behave as it used to.

The behavior of a MuJoCo-based balancing controller is shown in Figure 6.6. Notice that one of the legs oscillates with a higher magnitude. This is in accordance with our experience – one of the legs is stiffer to the point where it sometimes does not even extend when upside down. I tried to train new agents with different mean values of the parameters of the knees (recall that this value was also randomized) but did not see an improvement.

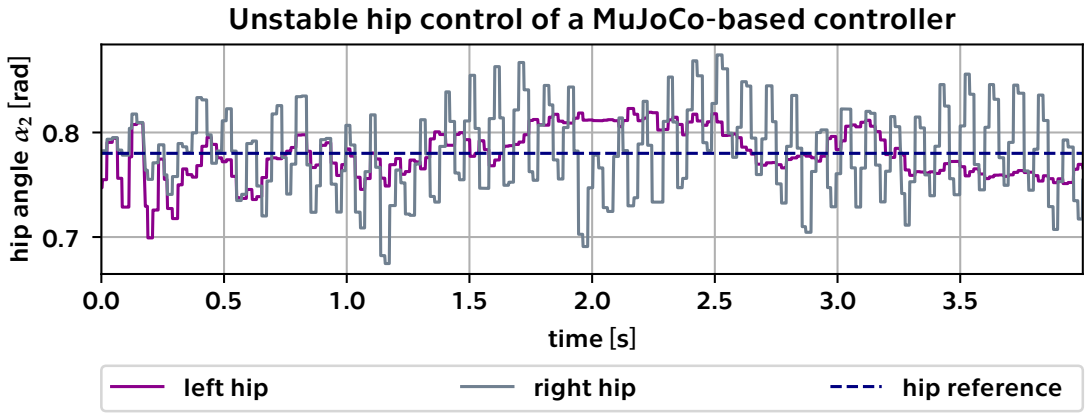


Figure 6.6: The hips were likely not correctly identified and the controller did not manage to maintain a fixed height.

Due to this issue, we did not perform any other experiments with hip control, both due to the dangers to the robot and because the oscillation visible in Figure 6.6 would make any velocity tracking unstable, even with a good wheel controller. Even though the main reason behind training these more complex agents was to allow the robot to adjust its roll angle, out of curiosity I also tested the performance of the agents with the control over hips handed over back to the motors. The results can be found in Figure 6.7.

An interesting and relatively straight-forward future endeavour could lie in simplifying the task as follows: instead of the agent directly controlling the torques, it could instead request their positions and the motor's PD controllers could set the torques appropriately. One might even argue that leaving the problem that is easily solvable by the motor controllers running at 20 kHz is even the proper engineering solution.

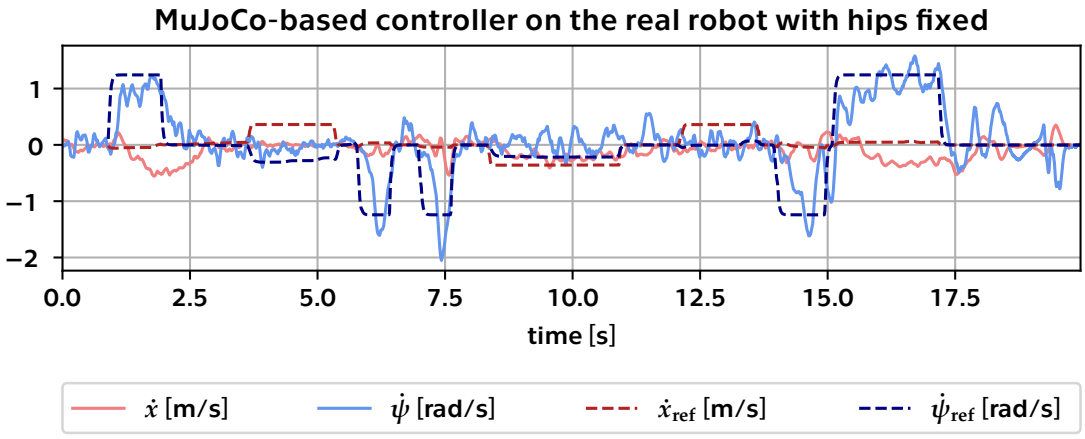


Figure 6.7: When hips of the MuJoCo based controller are fixed, the agent stabilizes the robot and can event rack yaw rate $\dot{\varphi}$ reference. However, it disregards the reference for forward velocity $\dot{x}$.

Chapter 7

Liability of Future AI Systems

In 2020, the European Comission released a white paper[1] on Artificial Intelligence, in which is states that it "entails a number of potential risks, such as opaque decision-making, gender-based or other kinds of discrimination, intrusion in our private lives or being used for criminal purposes" [42]. The stated purpose of the document is to define policies that will govern the use and research of AI for civil applications.

In the context of AI-system development, the most relevant part is chapter five, which sets out to define the regulatory framework for AI. It mentions several key requirements including safety, transparency, fairness and accountability to promote trust in the field. It has been shown that certain AI systems that are already deployed in practice can exhibit bias with regard to protected characteristics. As an example, [43] discusses COMPAS, a tool that forecasts the risk of recidivism of an individual and is used in courts in several jurisdictions in the USA, which has been shown to be biased with respect to gender and race.

For RL, which would typically be used in place of standard control system algorithms, liability (and, tangentially, safety certification) is arguably the most important aspect. In general, if a product is shown to be defective, the manufacturer is liable for any damanges under the Product Libaility Directive. However, as the white paper states, it might be difficult to judge whether an AI-based product is faulty – when a wheel falls off, the defect is clear but with black-box systems (which most deep AI models are), it is hard to show a causal link. The authors argue that current consumer protection legislature may not suffice and that it might even be necessary to shift the burden of proof on the manufacturer.

[1] https://eur-lex.europa.eu/EN/legal-content/glossary/white-paper.html

Additionally, the white paper proposes a few possible relevant requirements, such as:

- the need to provide assurances that the training datasets "cover all the relevant scenarios needed to avoid dangerous situations",
- proper documentation of the training procedures,
- possibility of human oversight (for instance a way to deactivate of the AI system in case of unsafe operation),
- robustness and accuracy of at least being able to correctly identify their levels.

Currently, there is legislation in progress to pass these results into a directive, which would then have to be implemented into each countries legislation [44]. Given the influence of the old continent, it is possible that such action would cause a domino effect among other democracies adopting similar measures.

7.1 Interpretability & Explainability

It would seem that at least in the EU, we will need a way to explain our (deep) models in the near future. This concerns the field of XAI, which is still relatively unknown. However, if a prediction made in 2021 by the consultancy firm Gartner is to be believed, XAI is expected to reach plateau of productivity within 5-10 years [45]. In the same year, they predicted chatbots to be productive in less than two years and I would argue that their assessment was correct, as they are already being incorporated in certain web search engines [46].

In this chapter, we will take a look at the current research being done in the field with particular considerations for RL. This could serve as a starting point for possible future applied research in RL, in case it is deemed necessary by the law-makers.

We will illustrate the techniques discussed on one of the models introduced in Section 4.3, namely an SAC agent trained to balance a Segway in ideal conditions. To obtain a dataset with approximately i.i.d. samples, the trained controller was left to interact with the environment for a million timesteps ($\geq 10^4$ episodes to promote variety in initial conditions). The resulting dataset was then subsampled to 10^4 observations to mitigate the correlation between samples – on average, this should result in only one sample per episode.

7.1.1 Model-agnostic Methods

Let us start with looking at the two terms in the title of this section. Interpretability and explainability are often used interchangeably and there are no widely accepted definitions [47]. One definition for both, given in [48], is "the degree to which an observer can understand the cause of a decision".

Some models are intrinsically interpretable, at least to an extent. An example of such models are decision trees, where the reasons for each concrete decision (this is known as local interpretability) can be understood analyzing the branching that lead to it. They are

also globally interpretable (though perhaps not fully), because we can look at the factors that influence the decisions the most (decision nodes closest to the root).

On the other hand, deep NNs have so many parameters and their interplay is so complex that a human cannot realistically comprehend them beyond a single layer with a handful of neurons. Therefore, to make sense of their outputs, we need a helping hand.

An excellent introductory book to this topic by Christoph Molnar [49], lists a wide variety of model-agnostic methods. That is, methods that can be applied to analyze any multivariable function such as deep NNs.

One possible approach is to only focus on a single feature at a time and analyze its effect on the output(s). For example, the Partial Dependence Plot (PDP) shows the marginal expected value of a function for all valid values of a parameter. In practice, to compute the partial dependence on e.g. $\varphi = 1$, we need to compute the sample average on a modified dataset in which the value of φ in all observations is replaced by 1.

In our case, we could expect that, on average, the magnitude of the wheel action increases with growing φ, as seen in Figure 7.1. While this kind of method might be useful for finding poorly explored parts of the observation space which could cause unexpected behaviours of the model, I do not think they have much chance at being accepted by a regulatory body, because it lacks the nuance to detect the interaction between parameters.

Another technique which might be of interest is finding a *global surrogate model*. This means that we train an intrinsically interpretable model to approximate the predictions. The problem in this case is clear – if the surrogate model provides accurate enough predictions, it is quite possible that we should not have used DL in the first place.

So what if we used a set of such surrogate models to approximate different parts of the observation space? One such method is called Local Interpretable Model-agnostic Explanations (LIME) [50]. Whenever we need to explain a particular output of our blackbox model, LIME fits a sparse linear model to the predictions of the blackbox model on the neigborhood of the point of interest. An example of this can be seen in Figure 7.2.

Another popular approach is known as SHAP values, introduced in [51]. As primer on the topic, I recommended reading the documentation of the Python `shap` packge.[2] The technique is based on Shapley values from game-theory, which are used for analyzing which subsets of players on a team are the most impactful for the outcome.

If a player's absence leads to a lower score of the team, they get a positive Shapley value. On the other hand, if they are detrimental to the result, they receive a negative value. In this case, the observation features are viewed as the players and the blackbox model as the game's outcome.

The above selection of methods, as well as many others, can be found in [49] along much more detail and examples. Even though these methods are general, or even focused on supervised learning, I believe that the local explanations might be able to satisfy the directive (in case of an accident) so I included them. Next, we take a look at some examples of how these methods were applied in RL and also introduce techniques which are more tailored towards it.

[2] https://shap.readthedocs.io/en/latest/overviews.html

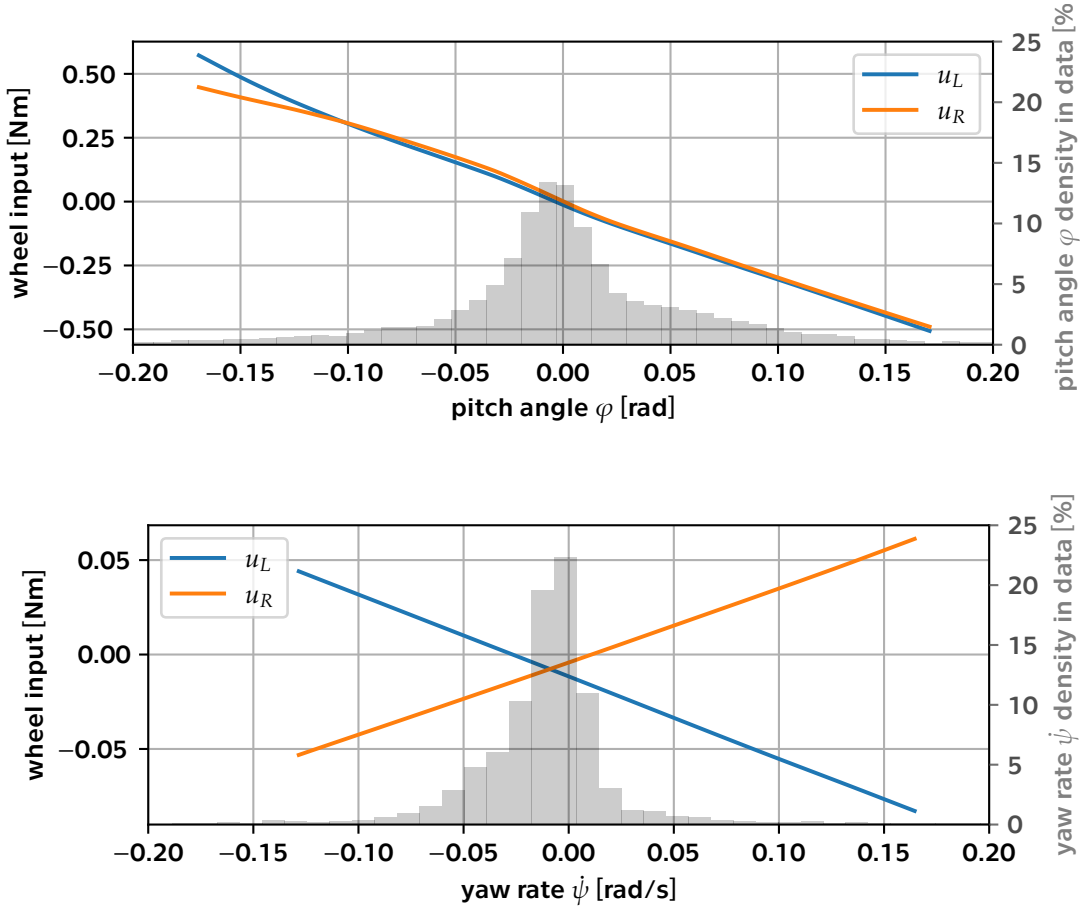


Figure 7.1: Partial dependence plot for Segway balancing. In particular, note that the expected action with zero pitch angle φ is nonzero and for negative values, the actions are not the same. This can possibly hint at issues with the model but it may be due to the data (especially the latter problem).

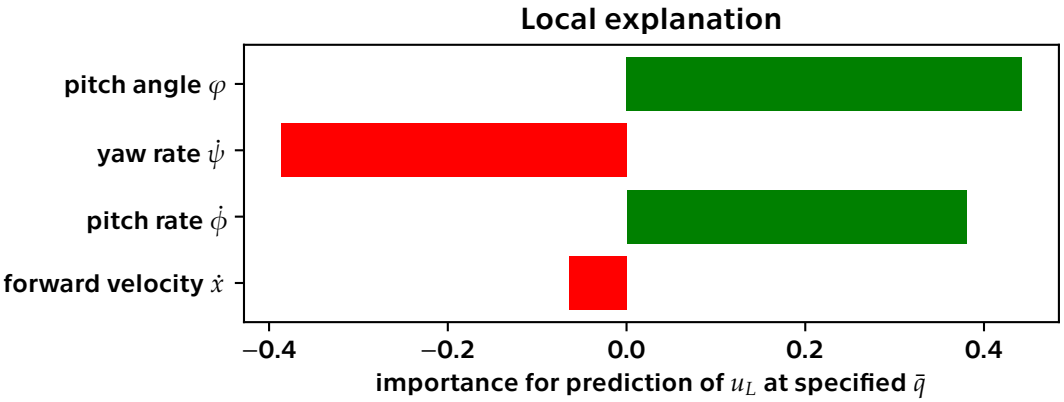


Figure 7.2: LIME explanation for left wheel action u_L in the case when the Segway is going forward, rotating slightly but at the same time, it is falling backwards, $\bar{q} = (0.1, -0.2, 0.5, -0.2)$. The $x-$axis corresponds to the effect the respective feature has on the model's prediction. Namely, it is the coefficient of a Ridge regressor fitted to the neigborhood of $\bar{q}$.

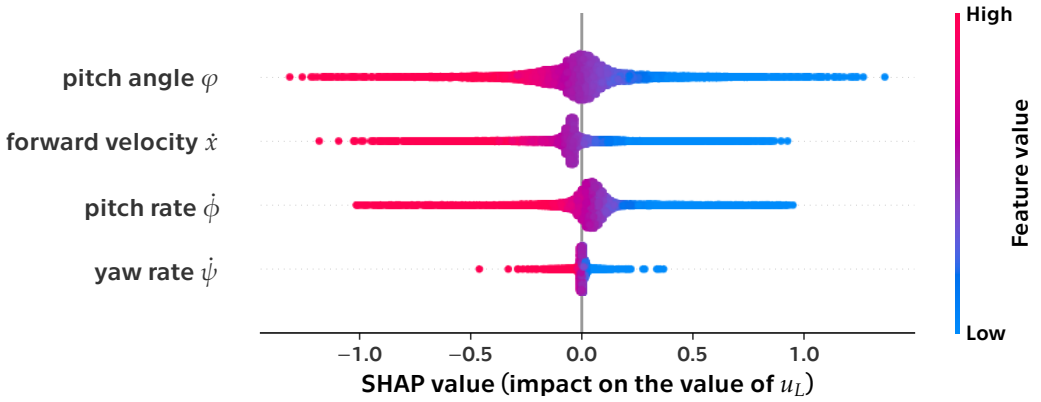


Figure 7.3: A beeswarm plot of SHAP values for left wheel control input u_L. Each point represents a single observation. Their colors correspond to the value of the respective feature (highly positive values in red, negative in blue) and their positions encode the impact they have on the action (the more right, the higher the requested torque).
We can see that in the dataset, the potentially most impactful variable was the pitch angle and that except for extremal values, yaw rate $\dot{\psi}$ only had mild impact, as all the intermediate purple colors are centered around the origin. Lastly, notice that intermediate values of $\dot{x}$ had a slightly negative impact on the prediction, which suggests a possible problem with the model or the dataset, as we would expect those to be around zero.

7.2 Explainability in Reinforcement Learning

When it comes to XAI in RL, the outlooks are largely the same as in the case of learning – most current research is done in supervised ML, as a very recent summary paper on the topic observes [47]. Nonetheless, progress in Explainable Reinforcement Learning (XRL) has been made in the past decade, despite the fact that RL breaks the common i.i.d. assumption, as we have discussed previously in Section 3.3.3. Let us now go over some of the interesting applications and results the overview mentions.

When the surrogate methods we mentioned in the previous section are applied to RL, they are often called *policy simplification* methods [47]. In [52], the authors show that when a DQN is trained on a fully observable Lava Gridworld (discrete observation and action space), one can create a set of rules that accurately describe the behavior of a "reasonable agent". This approach, however, might not be easily generalizable to continuous action and observation spaces.

In [53], we can see decision trees being constructed to explain a black box model. They introduce a novel algorithm tha strategically generates samples from the blackbox model to prevent overfitting. They have shown that their method can be applied to approximate RL agents trained for continuous action spaces.

If we are willing to structure our solution so that DL is only a single cog in a larger machine, an inspiration can be [54], where the authors develop what they call a *risk-aware* RL to control an autonomous vehicle. Their solution adds a collision prediction map generated by a classical model, which not only increases the performance of standard RL algorithms they tested but can be used in conjuction with post-hoc XAI to ensure that the chance of collision is taken into account.

Finally, let us discuss a techinque which is not post-hoc, so it is not immediatately

applicable to the models I have developed in this work. Additionally, this is an emerging area of research and as such has not seen much success in practice apart from toy datasets. Nonetheless, I personally find it elegant from a theoretical point of view, so I think it deserves its own section.

7.2.1 | Causal Learning

The idea behind statistical machine learning is to observe massive amounts of i.i.d. data from a joint probability distribution $p(x, y)$ and to find a function f (in the case of DL a neural network) that models $p(y \mid x)$ as $y = f(x)$. Ideally, this predictor will be able to correctly model the relationship even on unseen data. However, should the unseen examples be sampled from an altered distribution, the model may become arbitrarily inaccurate [55]. This can be exploited by malicious actors as famously demonstrated by [56].

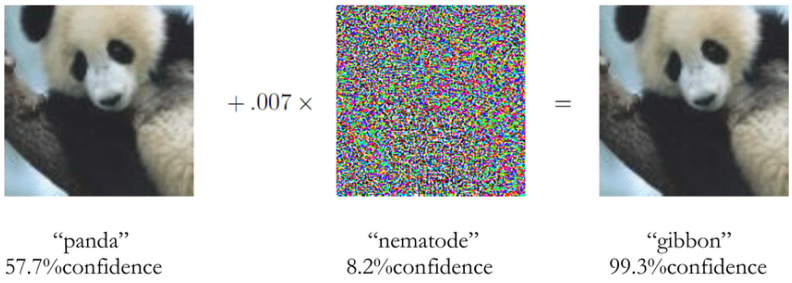


Figure 7.4: Illustration of an adversarial attack on a neural network. Image credit: Goodfellow et al. [56]

Additionally, this ignores possible causal relationships between the variables, which can be necessary for correct generalization of out-of-distribution examples, as noted by a review of causal representation learning, which is the main source of this section [55]. Having causal models can also allow the model to answer *interventional* ("Will SK8O fall if the pitch angle increases to 0.2 rad?") and *counterfactual* ("Would SK8O have fallen even if the pitch rate was negative?") questions.

For example, the amount of civil engineering doctorates awarded correlates with per capita consumption of mozzarella cheese with a correlation with $r = 0.96$.[3] Should the government subsidize cheese (more) in an effort to increase the number of highly-educated citizens, or rather should "big mozzarella" lobby for an increase in postgraduate students' funding? In this case, such questions may sound ludicrous but it is exactly the sort of interventional questions one may ask when deciding whether to run a discount campaign on their product.

Schölkopf et al. describe causal modeling as being between statistical machine learning and differential equations. It can benefit from data like ML (thus reducing the need for domain-expertise) but also provides a more in-depth understanding like the latter. Unfortunately, it typically requires the ability to perform *interventions* (see the effect of changing a particular feature) or data gathered in different environments [55].

[3] Courtesy of Tyler Vigen's famous Spurious Correlations (available at https://www.tylervigen.com/spurious-correlations)

Causal learning is based on the *Common Cause Principle*, which states that "if two observables X and Y are statistically dependent, then there exists a variable Z that causally influences both and explains all the dependence in the sense of making them independent when conditioned on Z" [55]. In the mozzarella example, Z could be the average household income, which allows more people to pursue higher studies and also buy more cheese.

The causal relationships can be modeled by directed (possibly acyclic) graphs, in which an edge $X \rightarrow Y$ exists if X is the cause of Y. When it comes to learning such structures from data, even identifying the components of the causal models is hard for both ML and humans. One technique for its discovery is showcased for example in [57], where they use a recurrent variational autoencoder, similar to the one we discussed in Section 3.4.1, with a special structure of the decoder, which internally learns the causal graph and achieves SOTA performance in medical timeseries generation.

Naturally, such solutions could bring a revolution for reinforcement learning. As [55] notes, learning invariances in a causal graph structure could lead to better generalization and possibly better sample efficiency, such as by creating an imagined world model (the authors go as far as to compare this to the way human reason about the world). They also mention that interventions may be necessary to discover causality – but that is the main principle of online RL. They stress that future RL should include this ability to formulate hypotheses in the imagined environment and then test them in the real environment.

The overview concludes by stating that here is still research to be done in this area not only in solving the problem but even in finding a good problem formulation.

7.3 Summary

This chapter was a notable digression from the main topic of the thesis. However, the topic of XAI is an important and fascinating one and is also clearly motivated by likely future legal requirements. I chose a lighter tone and focused primarily on the motivation for most of the discussion contained here as a way to balance the "legalese" that started it.

Chapter 8

Conclusions

We have reached the final chapter of this thesis, marking a moment to both reflect back on what we have seen and to envision potential avenues for further development.

The primary objective of this work was to develop a reinforcement learning-based controller for the SK8O robot which can not only prevent it from falling but also allow it to track a desired velocity reference. This was to be accomplished using two simulation models: a linear dynamical model covered in Chapter 4 and a full 3D model, which was implemented MuJoCo, as discussed in Chapter 5.

As outlined in Chapter 6, the resulting controllers were also successfully deployed onto the real robot. Remarkably, the agents trained on the simpler model managed to bridge the gap from simulation to the real world without any fine-tuning and in some regards exceed the performance of the classical controller currently used on the robot.

The policies trained using the full model, however, failed to stabilize SK8O, despite promising performance in simulation. It is possible that this outcome can be attributed to robot parameters having changed since their identification and the agents failing to generalize to the new system.

In Chapter 7, we took a slight detour and considered a possible upcoming EU liability directive. This motivated an introduction to the field of Explainable Artificial Intelligence and an overview of its most popular techniques. To maintain a connection to the rest of the thesis, I used a model developed previously as a recurring example.

To improve the presented outcomes, a new identification of the robot's parameters could be conducted, as they may have changed significantly since their original estimation. The improved model should then lead to agents better suited to control the real robot. Additionally, a conventional approach to augment the performance of RL agents is *fine-tuning* – the practice of continuing the training of the model's parameters based on the data collected from the real robot [58].

Over the course of the past year, many people have told me something along the lines that "reinforcement learning does not work", when they found out the topic of my thesis. However, I believe that at least in this case, it might be more accurate to say that though reinforcement learning can be finicky and requires a lot of experimenting, it can be *persuaded to* work in the end.

Appendix A

Contents of the Attachment

`text/`	this thesis in PDF format
`code/experiments/`	scripts used to generate the trajectories presented here
`code/explainability/`	scripts that generate the explainability
`code/training/`	scripts for training the agents
`models/`	some of the models presented in this thesis in ONNX format

Appendix B

Software

The codebase is developed in Python 3.9, documented with NumPy style docstrings and includes type hints. To install the dependencies, use the standard `requirements.txt` file. Due to the need to execute most training in the Research Center for Informatics, CTU Prague (RCI) cluster, there is also a Singularity container definition file `sk8o-rl.def`. One benefit of the Singularity container is that it supports headless OpenGL rendering via OSMesa, so that the script can save videos during training. Note that the processor architecture of the machine where the container is built has to be the same as the one where it is deployed. This was not an issue for me because the cluster has both AMD and Intel nodes.

Below, I will summarize the most significant (nonstandard) libraries and tools used to facilitate the development and training.

Reinforcement learning library

Having had worked with two of the most popular deep learning packages for Python, namely Tensorflow and PyTorch, I was knew which one I wanted to use. I find the latter to have better documentation and to be much more elegant and easy to use overall. This already narrowed down the choices somewhat, although there seems to be a trend among researchers to switch to PyTorch – none of the codes published in the recent papers I have seen used Tensorflow.

In the end, I settled on Stable Baselines 3 [59]. It is a mature package that contains all of the standard online RL algorithms and is extremely easy to start with. In particular, I use the 2.0 alpha version that supports the new open-sourced Python bindings for MuJoCo.

I used its implementation of the SAC algorithm and implemented the other two algorithms used in this thesis, D2RL and DroQ, as its modifications, so they were also largely based on the library implementation.

One issue I have encountered with this library was that it is not made to be modular (as a design choice), which complicates implementing new algorithms. For a future work, I believe the Tianshou[1] library, which strives to be more research-friendly, might be a good choice. However, at the time of writing, the package was not mature enough and I

[1] https://tianshou.readthedocs.io/

found its documentation to be lacking. For high-scale applications, Ray[rllib][2] might also be a good choice.

Experiment & Configuration management

For this thesis, thousands of training runs with many varying (hyper)parameters had to have been executed. Doing it manually at such scale is next to impossible and certainly a bad practice. There are three issues that need solving:

- merging default values, command-line arguments and system environment variables,
- generating reasonable sets of parameters,
- visualizing the results.

To solve the first problem, there is a wide variety of standard Python packages that parse command-line arguments, such as argparse[3] or click[4]. However, the number of parameters in this thesis is very high (>100) and their interplay is complex (different combinations of environments and tasks may lead to different default values). For a more fitting solution, I used hydra, developed by Meta (Facebook), which was created with deep learning in mind [60].

The second and third problems are solved jointly by Weights & Biases [61]. The sweep feature uses bayesian search to find a suitable combination of hyperparameters from a user-specified range. The web dashboard shows any logged data and even supports videos, which I found very useful when judging the performance of the algorithms.

[2] https://docs.ray.io/en/latest/rllib/index.html
[3] https://docs.python.org/3/library/argparse.html
[4] https://click.palletsprojects.com/en/8.1.x/

Appendix C

Equations of the Segway Model

When we discussed the linearized Segway model of SK8O in Chapter 4, I stated that the non-linear Lagrangian model for generalized coordinates $q_g = (x, \varphi, \psi)$ and input $u = (u_L, u_R)$ as seen in Figure C.1, is described by the equation

$$\ddot{q}_g = M(q_g)^{-1} \left(Bu - C(q_g, \dot{q}_g) - D\dot{q}_g - G(q_g) \right). \tag{C.1}$$

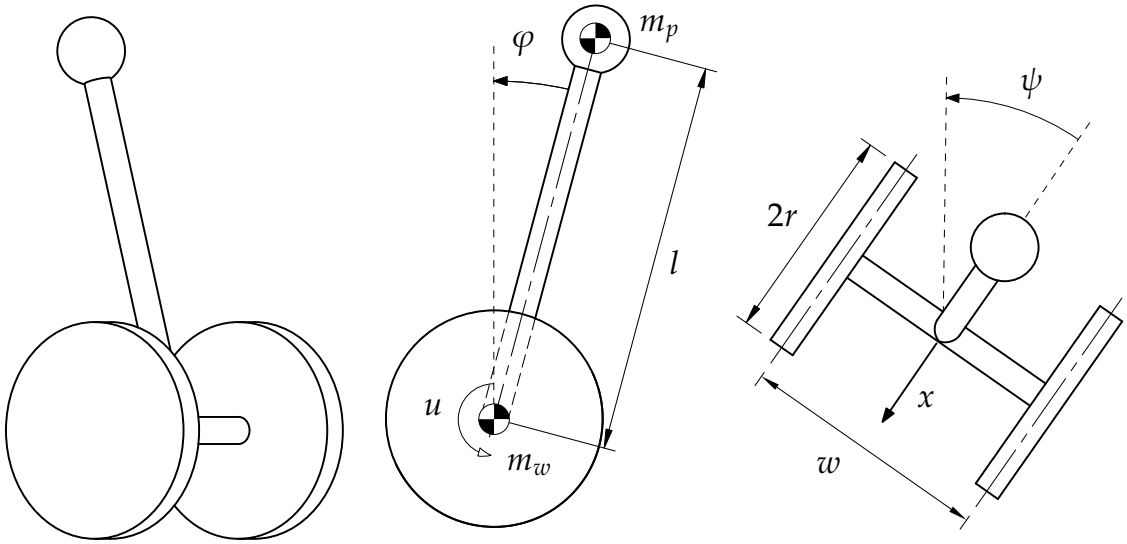


Figure C.1: The Segway model, its state and parameters

The values of the matrices are

$$M = \begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ 0 & 0 & m_{33} \end{bmatrix}, \quad C = \begin{bmatrix} 0 & c_{12} & c_{13} \\ 0 & 0 & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix},$$
$$D = \begin{bmatrix} d_{11} & d_{12} & 0 \\ d_{21} & d_{22} & 0 \\ 0 & 0 & d_{33} \end{bmatrix}, \quad B = \begin{bmatrix} 1/r & 1/r \\ -1 & -1 \\ -w/2r & w/2r \end{bmatrix}, \quad G = \begin{bmatrix} 0 \\ -mlg \sin\varphi \\ 0 \end{bmatrix}, \tag{C.2}$$

Symbol	Parameter	Value	Unit
b	damping coefficient	1×10^{-2}	$\mathrm{N\,m\,s\,rad^{-1}}$
J	wheel moment of inertia about its turning axis	7.35×10^{-4}	$\mathrm{kg\,m^2}$
K	wheel moment of inertia about the vertical axis	3.9×10^{-4}	$\mathrm{kg\,m^2}$
m_w	wheel mass	3×10^{-1}	kg
r	wheel radius	8×10^{-2}	m
w	distance between wheels	2.9×10^{-1}	m
l	COM height	2.907×10^{-1}	m
I_{px}	roll moment of inertia of the pendulum	1.5625×10^{-2}	$\mathrm{kg\,m^2}$
I_{py}	pitch moment of inertia of the pendulum	$1.186\,25 \times 10^{-2}$	$\mathrm{kg\,m^2}$
I_{pz}	yaw moment of inertia of the pendulum	$1.186\,25 \times 10^{-2}$	$\mathrm{kg\,m^2}$
m_p	mass of the Segway without wheels	4	kg

Table C.1: Parameters of the Segway model. All moments of inertia are shown with respect to the COM of the body.

where the elements are

$$\begin{gathered}
m_{11} = m_p + 2m_w + 2\frac{J}{r^2}, \quad m_{12} = m_{21} = m_p l \cos\varphi, \quad m_{22} = I_{py} + m_p l^2, \\
m_{33} = I_{pz} + 2K + \left(m_w + \frac{J}{r^2}\right)\frac{w^2}{2} - (I_{pz} - I_{px} - m_p l^2)\sin^2\varphi, \\
c_{12} = -m_p l\dot{\varphi}\sin\varphi, \quad c_{13} = m_p l\dot{\psi}\sin\varphi, \quad c_{23} = (I_{pz} - I_{px} - m_p l^2)\dot{\psi}\sin\varphi\cos\varphi, \\
c_{31} = m_p l\dot{\psi}\sin\varphi, \quad c_{32} = -c_{23}, \quad c_{33} = -(I_{pz} - I_{px} - m_p l^2)\dot{\varphi}\sin\varphi\cos\varphi, \\
d_{11} = \frac{2b}{r^2}, \quad d_{12} = d_{21} = -\frac{2b}{r}, \quad d_{22} = 2b, \quad d_{33} = \frac{w^2}{2r^2}b.
\end{gathered} \tag{C.3}$$

The values of the physical parameters can be found in Table C.1.

Appendix D

MuJoCo Model Parameters

Here I present the values of all the parameters that define SK8O, as shown in Figure D.1. The values are utilized in the MuJoCo model which can be found in the attachment in `code/training/env/sk8o_full/full_model.xml`.

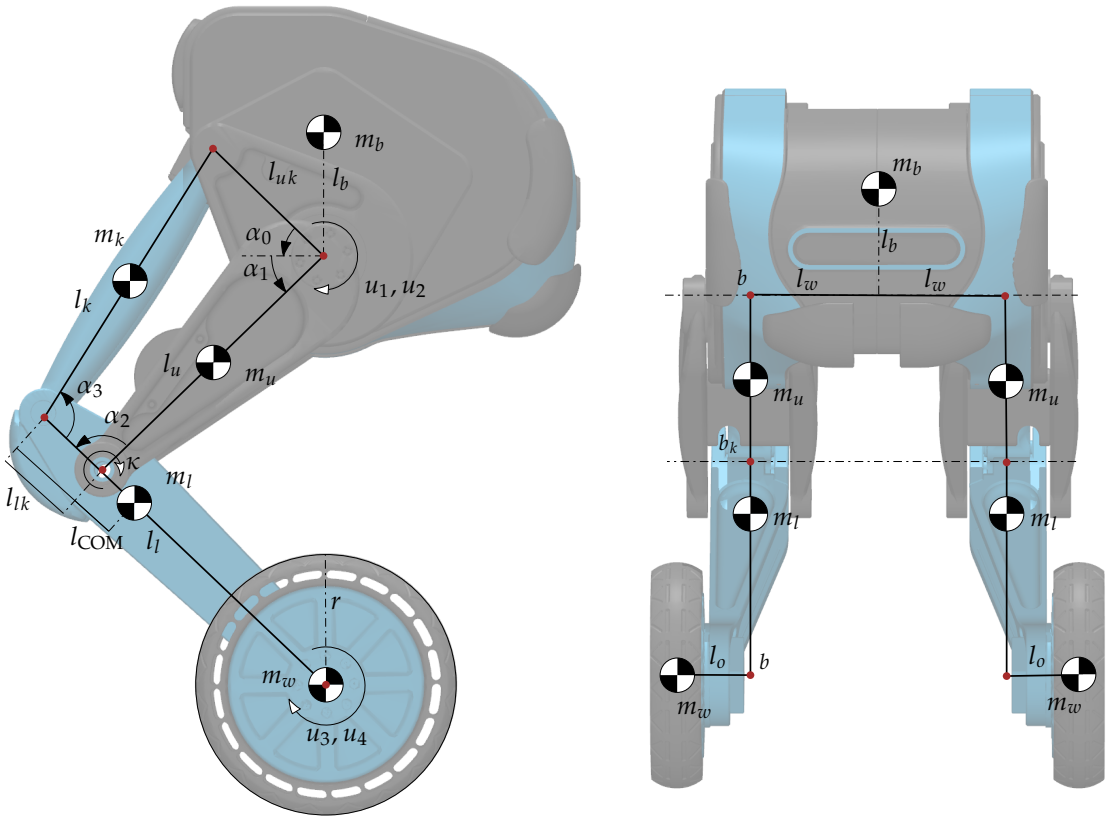


Figure D.1: Diagram of SK8O's structure superimposed over the 3D mesh

Body	Symbol	Value	Unit
-	b	0.01	$\mathrm{N\,m\,s\,rad^{-1}}$
body	m_b	3	kg
	l_b	100	mm
	l_{uk}	93	mm
	l_w	92.5	mm
upper leg	m_u	0.2	kg
	l_u	188	mm
lower leg	m_l	0.2	kg
	l_l	190	mm
	l_{lk}	50	mm
	l_{COM}	80	mm
	b_k	0.15	$\mathrm{N\,m\,s\,rad^{-1}}$
	κ	0.15	$\mathrm{N\,m\,rad^{-1}}$
	$\alpha_{2,\mathrm{ref}}$	-240	deg
kinematic loop	m_k	0.1	kg
	l_k	193	mm
wheel	m_w	0.3	kg
	l_o	25.6	mm
	r	80	mm

Table D.1: Parameters of the full model

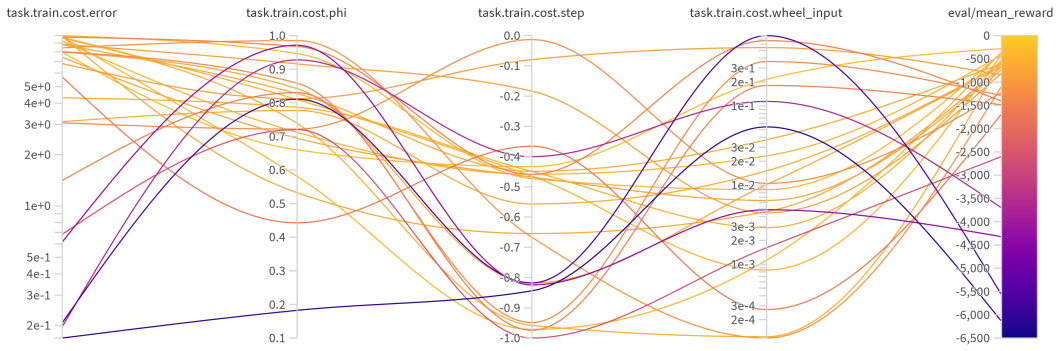
Body	J_{xx} [kg m^2]	J_{yy} [kg m^2]	J_{zz} [kg m^2]
body	9.25×10^{-2}	6.57×10^{-2}	6.57×10^{-2}
upper leg	6.04×10^{-4}	1.67×10^{-5}	5.91×10^{-4}
lower leg	9.75×10^{-4}	1.67×10^{-5}	9.62×10^{-4}
kinematic loop	6.36×10^{-4}	1.67×10^{-5}	6.22×10^{-4}
wheel	3.90×10^{-4}	3.90×10^{-4}	7.35×10^{-4}

Table D.2: Moments of inertia with respect to the COMs of all the bodies in the full model.

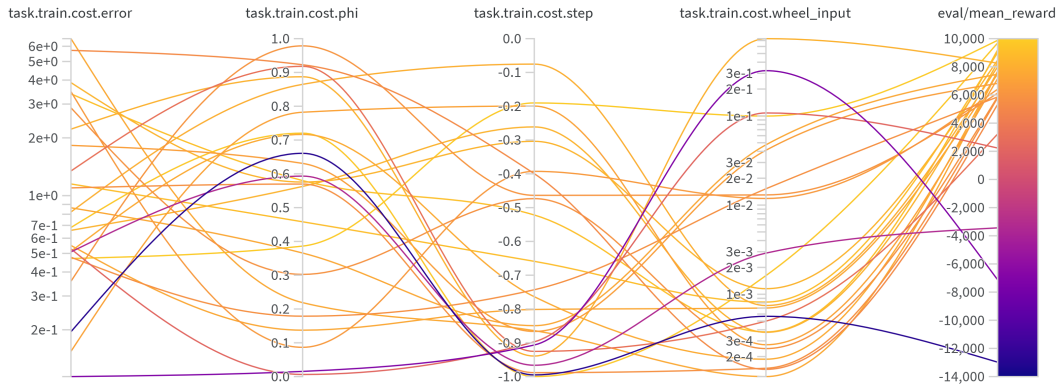
Appendix E

Reward Tuning

In sections 4.3.1 and 5.3.3, we have searched for a reasonable set of coefficients in the training reward function. The results were then shown in tables 4.2, 4.4. In Figure E.1, you can find all the combinations of parameters that were tested by Weights and Biases's sweep function and their results. The plots were exported from the Weights and Biases UI.



(**a**) Reward sweep for Segway balancing



(**b**) Reward sweep for Segway velocity tracking

Figure E.1: Reward sweeps used to find the rewards functions for training of RL agents. Each line corresponds to one training run with a set of parameters determined by the locations at which it intersects the vertical lines. Its color encodes the mean evaluation reward.

Appendix F

Director's Cut

Below, you can find attached data from training runs which were referenced from the main text but were not key to the understanding of the narrative, so they were moved here in an effort to increase the text's cohesion.

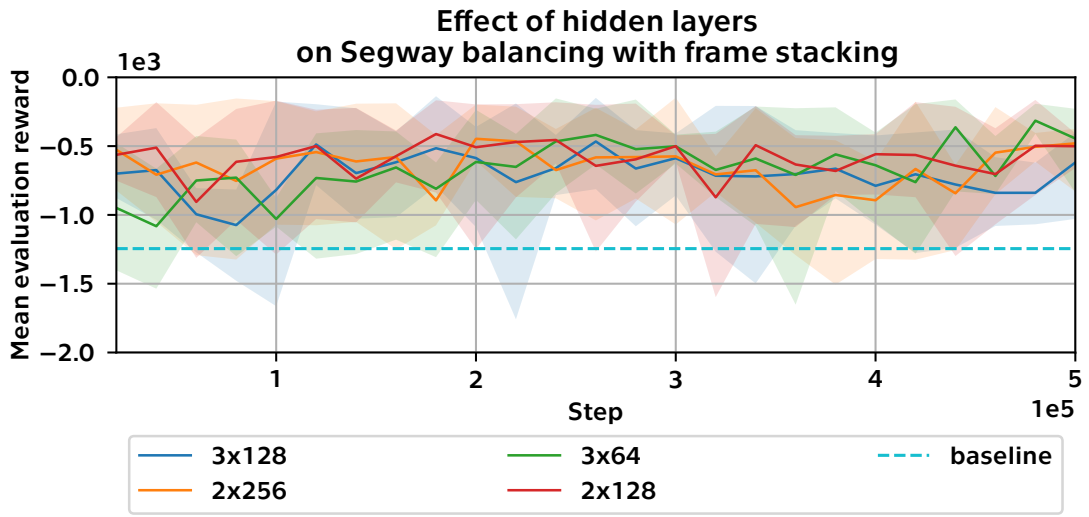


Figure F.1: The choice of hidden layers does not make a significant difference in evaluation reward, discussed in Section 4.3.2.

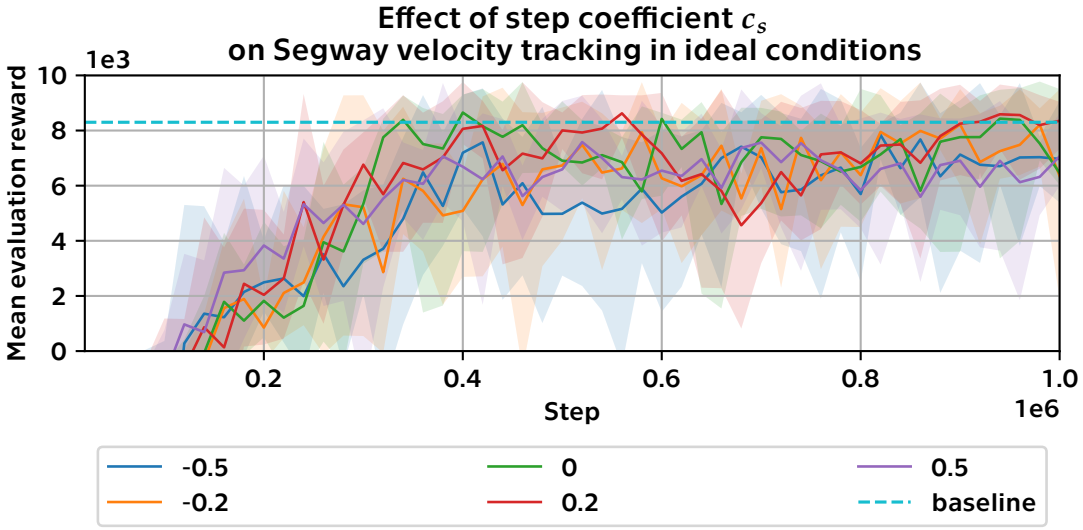


Figure F.2: Removing the step reward lowers variance at the cost of a slightly worse performance, so in Section 4.4.1, $c_s = 0$ was selected.

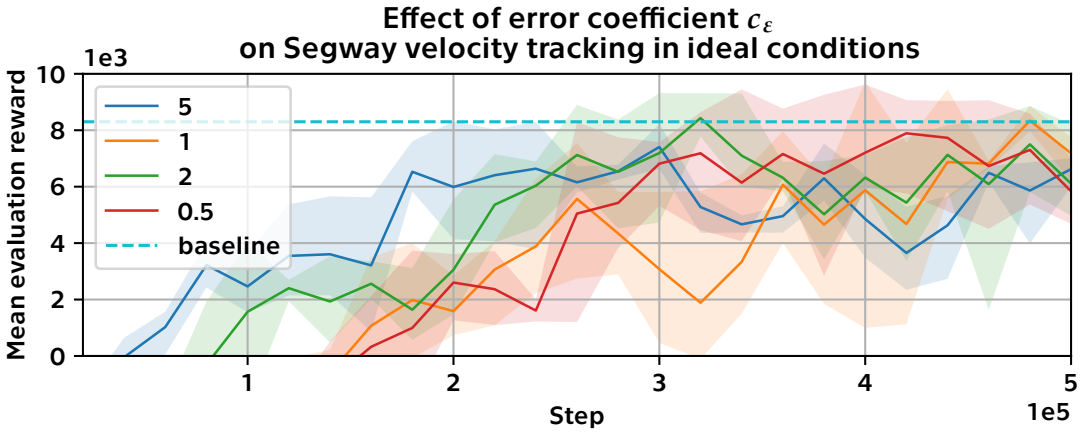


Figure F.3: Even though higher values of c_ε lead to faster learning initially, the value found by the sweep achieves better results in the end, discussed in Section 4.4.1.

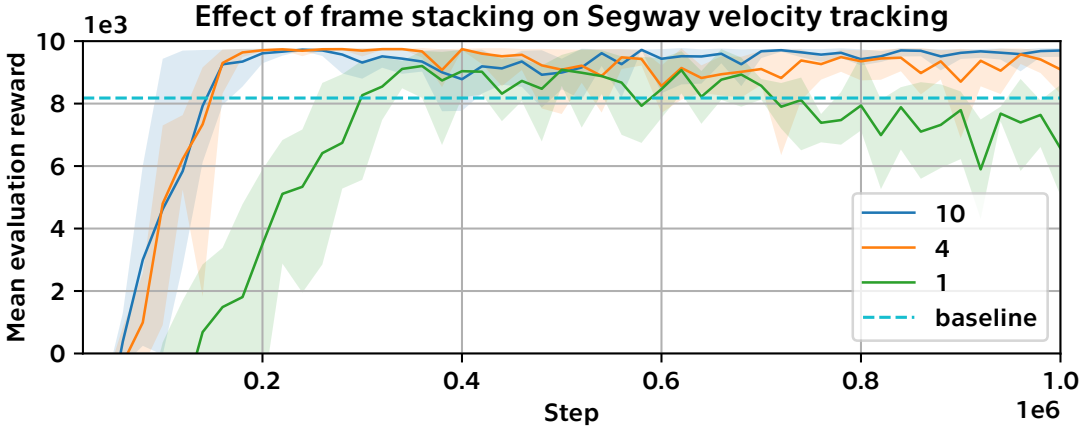


Figure F.4: The benefits of frame stacking are comparable to those found in balancing, as mentioned in Section 4.4.3.

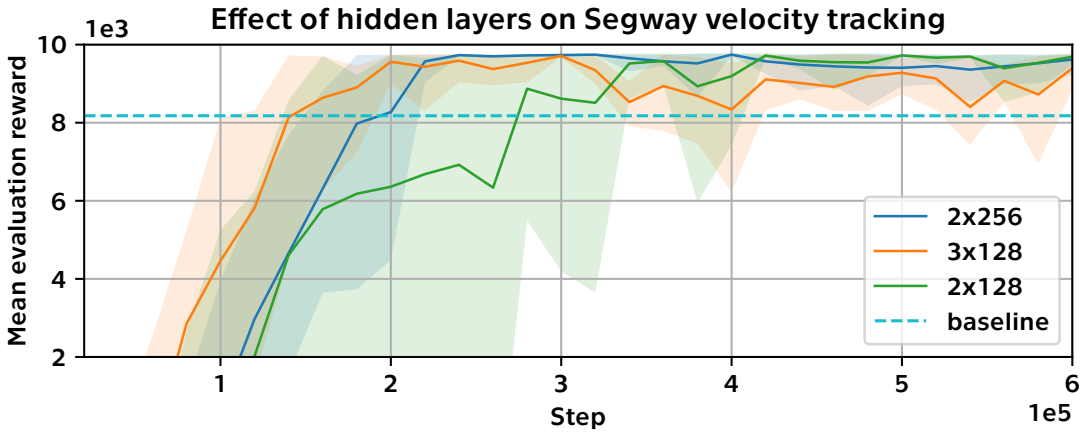


Figure F.5: The choice of architecture does not have a significant impact on prediction quality, although the deeper network may be slightly overfitted, discussed in Section 4.4.3.

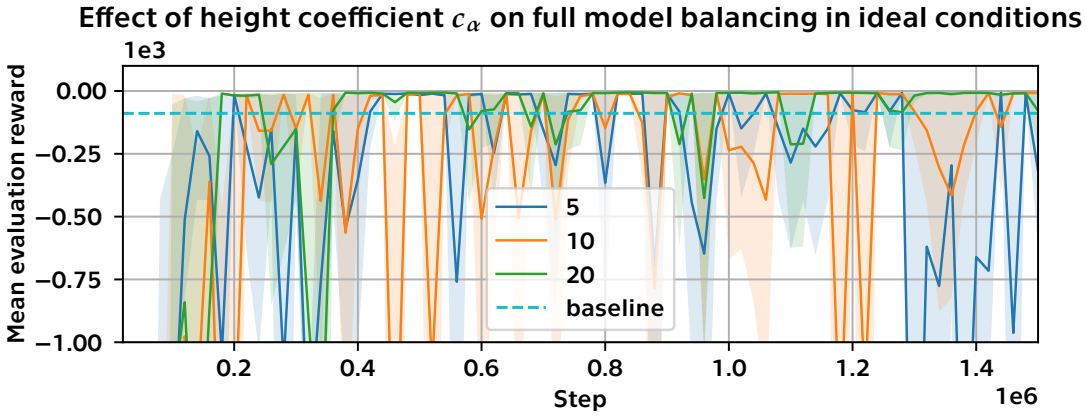


Figure F.6: Although with $c_\alpha = 20$, the agent takes longer to train, it performs more consistently in the end. Motivates the choice of the parameter in Section 5.3.3.

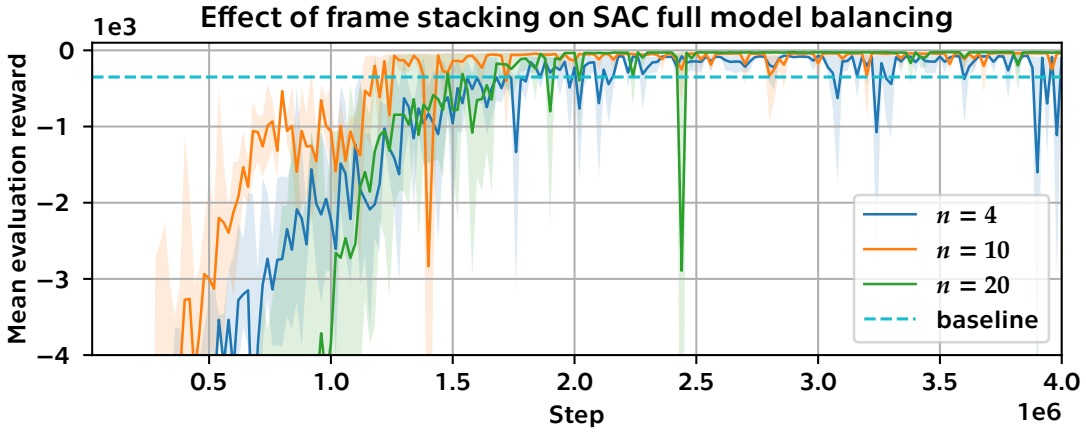


Figure F.7: In Section 5.3.7, I mention that D2RL outperforms SAC. For reference, the performance of SAC is included here too.

Bibliography

[1] *Pause Giant AI Experiments: An Open Letter*, en-US. [Online]. Available: https://futureoflife.org/open-letter/pause-giant-ai-experiments/ (visited on 04/12/2023).

[2] A. Kollarčík, "Modeling and control of two-legged wheeled robot," Diploma thesis, Czech Technical University in Prague, 2021.

[3] T. Bártík, "Autonomous sk8o robot," Bachelor thesis, Czech Technical University in Prague, 2022.

[4] L. Smith, I. Kostrikov, and S. Levine, *A Walk in the Park: Learning to Walk in 20 Minutes With Model-Free Reinforcement Learning*, en, arXiv:2208.07860 [cs], Aug. 2022. (visited on 03/02/2023).

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[6] T. Haarnoja, A. Zhou, K. Hartikainen, *et al.*, "Soft Actor-Critic Algorithms and Applications," *arXiv:1812.05905 [cs, stat]*, Jan. 2019, arXiv: 1812.05905. (visited on 03/17/2020).

[7] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, *Learning to Walk via Deep Reinforcement Learning*, arXiv:1812.11103 [cs, stat], Jun. 2019. DOI: 10.48550/arXiv.1812.11103. (visited on 09/27/2022).

[8] "Benchmark — Tianshou 0.5.0 documentation." (), [Online]. Available: https://tianshou.readthedocs.io/en/master/tutorials/benchmark.html (visited on 03/13/2023).

[9] F. Helfenstein, "Benchmarking deep reinforcement learning algorithms," 2021.

[10] C. Igoe, *Choosing target entropy; for soft-actor-critic (sac) algorithm*, Cross Validated, URL:https://stats.stackexchange.com/q/568751 (version: 2022-03-22). [Online]. Available: https://stats.stackexchange.com/q/568751.

[11] H. Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23, Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[12] X. Chen, C. Wang, Z. Zhou, and K. Ross, *Randomized Ensembled Double Q-Learning: Learning Fast Without a Model*, arXiv:2101.05982 [cs], Mar. 2021. DOI: 10.48550/arXiv.2101.05982. (visited on 03/08/2023).

[13] T. Hiraoka, T. Imagawa, T. Hashimoto, T. Onishi, and Y. Tsuruoka, *Dropout Q-Functions for Doubly Efficient Reinforcement Learning*, arXiv:2110.02034 [cs], Mar. 2022. DOI: 10.48550/arXiv.2110.02034. (visited on 03/08/2023).

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014, ISSN: 1533-7928. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html (visited on 05/12/2023).

[15] Y. Wu, X. Chen, C. Wang, Y. Zhang, and K. W. Ross, *Aggressive Q-Learning with Ensembles: Achieving Both High Sample Efficiency and High Asymptotic Performance*, arXiv:2111.09159 [cs], Nov. 2022. DOI: 10.48550/arXiv.2111.09159. (visited on 04/16/2023).

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," en, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, Number: 6088 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/323533a0. [Online]. Available: https://www.nature.com/articles/323533a0.

[17] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention Is All You Need," *arXiv:1706.03762 [cs]*, Dec. 2017, arXiv: 1706.03762. (visited on 03/27/2020).

[18] S. Sinha, H. Bharadhwaj, A. Srinivas, and A. Garg, *D2RL: Deep Dense Architectures in Reinforcement Learning*, arXiv:2010.09163 [cs], Nov. 2020. DOI: 10.48550/arXiv.2010.09163. (visited on 09/27/2022).

[19] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3803–3810, May 2018, arXiv: 1710.06537. DOI: 10.1109/ICRA.2018.8460528. (visited on 03/15/2020).

[20] X. Chen, A. Ghadirzadeh, J. Folkesson, and P. Jensfelt, *Deep Reinforcement Learning to Acquire Navigation Skills for Wheel-Legged Robots in Complex Environments*, arXiv:1804.10500 [cs, stat], Apr. 2018. DOI: 10.48550/arXiv.1804.10500. (visited on 09/27/2022).

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, Jul. 2019, arXiv: 1509.02971. (visited on 03/02/2020).

[22] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, *Hindsight Experience Replay*, arXiv:1707.01495 [cs], Feb. 2018. DOI: 10.48550/arXiv.1707.01495. (visited on 02/07/2023).

[23] OpenAI, *Ingredients for robotics research*, en-US. [Online]. Available: https://openai.com/research/ingredients-for-robotics-research (visited on 05/04/2023).

[24] A. Y. Ng, D. Harada, and S. J. Russell, "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping," in *Proceedings of the Sixteenth International Conference on Machine Learning*, ser. ICML '99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Jun. 1999, pp. 278–287, ISBN: 978-1-55860-612-8. (visited on 04/23/2023).

[25] Q. Zhang, W. Pan, and V. Reppa, *Model-Reference Reinforcement Learning Control of Autonomous Surface Vehicles with Uncertainties*, arXiv:2003.13839 [cs, eess, math], Mar. 2020. DOI: 10.48550/arXiv.2003.13839. (visited on 04/19/2023).

[26] A. Levy, G. Konidaris, R. Platt, and K. Saenko, *Learning Multi-Level Hierarchies with Hindsight*, arXiv:1712.00948 [cs], Sep. 2019. DOI: 10.48550/arXiv.1712.00948. (visited on 02/07/2023).

[27] S. Ross, G. J. Gordon, and J. A. Bagnell, *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*, en, arXiv:1011.0686 [cs, stat], Mar. 2011. (visited on 05/01/2023).

[28] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 661–668. [Online]. Available: https://proceedings.mlr.press/v9/ross10a.html.

[29] J. Ho and S. Ermon, *Generative Adversarial Imitation Learning*, arXiv:1606.03476 [cs], Jun. 2016. DOI: 10.48550/arXiv.1606.03476. (visited on 05/01/2023).

[30] L. Xie, S. Wang, S. Rosa, A. Markham, and N. Trigoni, *Learning with Training Wheels: Speeding up Training with a Simple Controller for Deep Reinforcement Learning*, arXiv:1812.05027 [cs], Dec. 2018. DOI: 10.48550/arXiv.1812.05027. (visited on 09/27/2022).

[31] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering Atari, Go, chess and shogi by planning with a learned model," en, *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 2020, Number: 7839 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. [Online]. Available: https://www.nature.com/articles/s41586-020-03051-4 (visited on 05/09/2023).

[32] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, *Mastering Atari with Discrete World Models*, arXiv:2010.02193 [cs, stat], Feb. 2022. DOI: 10.48550/arXiv.2010.02193. (visited on 05/10/2023).

[33] V. Micheli, E. Alonso, and F. Fleuret, *Transformers are Sample-Efficient World Models*, arXiv:2209.00588 [cs], Mar. 2023. DOI: 10.48550/arXiv.2209.00588. (visited on 04/12/2023).

[34] M. Janner, J. Fu, M. Zhang, and S. Levine, *When to Trust Your Model: Model-Based Policy Optimization*, arXiv:1906.08253 [cs, stat], Nov. 2021. DOI: 10.48550/arXiv.1906.08253. (visited on 03/13/2023).

[35] W. van Heeswijk, *Why Hasn't Reinforcement Learning Conquered The World (Yet)?* en, Nov. 2021. [Online]. Available: https://towardsdatascience.com/why-hasnt-reinforcement-learning-conquered-the-world-yet-459ae99982c6 (visited on 04/09/2023).

[36] D. Han, K. Doya, and J. Tani, *Variational Recurrent Models for Solving Partially Observable Control Tasks*, arXiv:1912.10703 [cs, eess, stat], Dec. 2019. DOI: 10.48550/arXiv.1912.10703. (visited on 02/07/2023).

[37] S. Kim and S. Kwon, "Dynamic modeling of a two-wheeled inverted pendulum balancing mobile robot," *International Journal of Control, Automation and Systems*, vol. 13, Aug. 2015. DOI: 10.1007/s12555-014-0564-8.

[38] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "OpenAI Gym," *arXiv:1606.01540 [cs]*, Jun. 2016, arXiv: 1606.01540. (visited on 04/29/2020).

[39] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[40] DeepMind, *Open-sourcing MuJoCo*, en, https://www.deepmind.com/blog/open-sourcing-mujoco, [Online; accessed 2023/31/01]. (visited on 01/31/2023).

[41] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, *Time Limits in Reinforcement Learning*, arXiv:1712.00378 [cs], Jan. 2022. (visited on 10/17/2022).

[42] E. Commission, *White Paper on Artificial Intelligence: A European approach to excellence and trust*, en. [Online]. Available: https://commission.europa.eu/publications/white-paper-artificial-intelligence-european-approach-excellence-and-trust_en (visited on 04/27/2023).

[43] Q. Zhou, J. Marecek, and R. N. Shorten, *Fairness in Forecasting of Observations of Linear Dynamical Systems*, arXiv:2209.05274 [cs, eess, math, stat], Sep. 2022. DOI: 10.48550/arXiv.2209.05274. (visited on 02/02/2023).

[44] *Liability Rules for Artificial Intelligence*, en. [Online]. Available: https://commission.europa.eu/business-economy-euro/doing-business-eu/contract-rules/digital-contracts/liability-rules-artificial-intelligence_en (visited on 04/27/2023).

[45] *The 4 Trends That Prevail on the Gartner Hype Cycle for AI, 2021*, en, Jul. 2021. [Online]. Available: https://www.gartner.com/en/articles/the-4-trends-that-prevail-on-the-gartner-hype-cycle-for-ai-2021 (visited on 05/13/2023).

[46] *Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web*, en-US, Feb. 2023. [Online]. Available: https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/ (visited on 05/13/2023).

[47] A. Krajna, M. Brcic, T. Lipic, and J. Doncevic, *Explainability in reinforcement learning: Perspective and position*, arXiv:2203.11547 [cs], Mar. 2022. (visited on 12/22/2022).

[48] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," en, *Artificial Intelligence*, vol. 267, pp. 1–38, Feb. 2019, ISSN: 0004-3702. DOI: 10.1016/j.artint.2018.07.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370218305988 (visited on 05/13/2023).

[49] C. Molnar, *Interpretable Machine Learning, A Guide for Making Black Box Models Explainable*, 2nd ed. 2022. [Online]. Available: https://christophm.github.io/interpretable-ml-book.

[50] M. T. Ribeiro, S. Singh, and C. Guestrin, *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*, arXiv:1602.04938 [cs, stat], Aug. 2016. DOI: 10.48550/arXiv.1602.04938. (visited on 05/13/2023).

[51] S. Lundberg and S.-I. Lee, *A unified approach to interpreting model predictions*, 2017. arXiv: 1705.07874 [cs.AI].

[52] J. McCarthy, R. Nair, E. Daly, R. Marinescu, and I. Dusparic, *Boolean Decision Rules for Reinforcement Learning Policy Summarisation*, arXiv:2207.08651 [cs], Jul. 2022. DOI: 10.48550/arXiv.2207.08651. (visited on 05/10/2023).

[53] O. Bastani, C. Kim, and H. Bastani, *Interpreting Blackbox Models via Model Extraction*, en, arXiv:1705.08504 [cs], Jan. 2019. (visited on 05/08/2023).

[54] E. Candela, O. Doustaly, L. Parada, F. Feng, Y. Demiris, and P. Angeloudis, "Risk-aware controller for autonomous vehicles using model-based collision prediction and reinforcement learning," en, *Artificial Intelligence*, vol. 320, p. 103 923, Jul. 2023, ISSN: 0004-3702. DOI: 10.1016/j.artint.2023.103923. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370223000693 (visited on 04/28/2023).

[55] B. Schölkopf, F. Locatello, S. Bauer, *et al.*, *Towards Causal Representation Learning*, arXiv:2102.11107 [cs], Feb. 2021. DOI: 10.48550/arXiv.2102.11107. (visited on 02/01/2023).

[56] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and Harnessing Adversarial Examples*, arXiv:1412.6572 [cs, stat], Mar. 2015. DOI: 10.48550/arXiv.1412.6572. (visited on 05/20/2023).

[57] H. Li, S. Yu, and J. Principe, *Causal Recurrent Variational Autoencoder for Medical Time Series Generation*, arXiv:2301.06574 [cs, eess], Jan. 2023. DOI: 10.48550/arXiv.2301.06574. (visited on 04/12/2023).

[58] R. Julian, B. Swanson, G. S. Sukhatme, S. Levine, C. Finn, and K. Hausman, *Never Stop Learning: The Effectiveness of Fine-Tuning in Robotic Reinforcement Learning*, en, arXiv:2004.10190 [cs, stat], Jul. 2020. (visited on 05/24/2023).

[59] A. Hill, A. Raffin, M. Ernestus, *et al.*, *Stable baselines*, https://github.com/hill-a/stable-baselines, 2018.

[60] O. Yadan, *Hydra - a framework for elegantly configuring complex applications*, Github, 2019. [Online]. Available: https://github.com/facebookresearch/hydra.

[61] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020. [Online]. Available: https://www.wandb.com/.