TUM School of Computation, Information and Technology
Technische Universität München

Enabling Reinforcement Learning on Real Robots

Antonin J. Raffin

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Achim Lilienthal

Prüfende der Dissertation:
1. Prof. Dr. Alin Albu-Schäffer
2. Assoc. Prof. Dr. Jens Kober
3. Prof. Dr. Majid Khadiv

Die Dissertation wurde am 12.12.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 20.05.2025 angenommen.

Preface

This dissertation is based on research undertaken at the Institute of Robotics and Mechatronics of German Aerospace Center (DLR) in Oberpfaffenhofen, Germany. It took six years (2018-2024) to accumulate the results which are reported in the thesis.

First, I would like to thank Freek, who introduced me to RL and later welcomed me to DLR, always being supportive. This thesis would not have been possible without the guidance of João and Alin, and the invaluable feedback I received from Jens and Olivier.

I would also like to express my gratitude to Rico, Sebastian and Oliver, who introduced me to the field of reinforcement learning research while I was an intern at TU Berlin. To my students, Noah, Megan and Andres, who enabled me to explore other areas of RL.

I should also mention my former ENSTA colleagues, notably Tim, Flo, René, Thibault and David, who were there when we started Stable Baselines. To the ENSTAR members Dara, Yvon, Agathe, Antoine and many others, with whom I discovered robotics.

Talking about SB2 and SB3, I learned a lot with all the maintainers, Anssi, Quentin, Adam, Max and Ashley. I was also grateful to collaborate with Costa, Bas and Jelle on RL research.

A special thank you to my close friends who have been there for years: Léa, Jean-Baptiste, Fanny and Cecilia. Special thanks to Lucille-Marie, who believed in me (yes, I finally made it!).

À ce propos, je tiens tout particulièrement à remercier la TUM, qui a fait preuve d'excellence sur le plan administratif durant ces nombreuses années. Elle n'a jamais failli à me montrer que l'humain primait toujours sur les procédures et les cases à cocher.

Next come all my friends and colleagues, with whom I enjoyed spending time, discussing, bouldering, dancing, ... making the doctorate a nice journey. Thank you Martin, Max, Elli, Matthias, Posi, Sebi, Gabriel, Lydia, Irene, Karen, Caspar, Johannes, Manish, Olivia, Ribin, Xuming, Ana, Juliane, Eva, Konrad, Alison, Camille, Tiphaine, Eugénie and Sergio.

This thesis would not have been possible without the support of the David and Bert team, who helped me work on and repair the robots. Many other DLR colleagues also helped me along the way, and I enjoyed discussing things with them during breaks. So, in random order, thank you Seba, Maged, Timo, Markus, Marie, Sam, Adrien, Marco, Abhishek, Leon, Mathilde, Jongsoek, Klaus, German Antonin, Flo, Lukas, Seba, Jinoh, Milan, Daniel, Annika, Davide, Tristan, Xiaozhou, Margherita, Neal, Basti, Martin (x2), Manuel (x2), Max (x2), Viktor, Thomas and Henry, as well as anyone else I may have forgotten.

Coming to the end, I would like to thank my family and my parents for their constant support during my studies and after.

അവസാനമായി, എന്റെ അവസാന വാക്കുകൾ എന്റെ (ട്രിപ്പിൾ) ഭാര്യ റിയയുടേതാണ്, അവർ എന്റെ ജീവിതം സന്തോഷകരമാക്കുന്നു.

Abstract

The main goal of this thesis is to enable robots to learn controllers directly on real hardware in a short time using deep reinforcement learning (DRL).

DRL excels in simulation, usually starting from scratch, and impressive results can even be transferred to real robots. However, successful transfers come at the cost of extensive domain randomization, time-consuming reward engineering, and tedious modeling.

Learning on real hardware eliminates the gap between simulation and reality, but introduces new challenges such as ensuring the safety of the robot and its environment, the inability to parallelize training, time delays, broken actuators, and manual resets.

To overcome these challenges, we investigate how to empower the learning agent with expert knowledge, saving costly interactions with the real world. To apply DRL directly on real robots, we propose a smoother exploration scheme, a safer alternative to the standard step-based exploration. We show that this new exploration enables learning on the robot and that there is a trade-off between smoothness and performance.

We use different types of prior knowledge to reduce training time and improve performance on two robots with elastic components, the *David* elastic neck and the *bert* quadruped. A data-driven fault-tolerant pose estimator is presented and reused to learn a controller, by providing a feedforward component.

To learn locomotion policies, we contribute an open-loop baseline consisting of simple oscillators. This baseline highlights the current limitations of DRL algorithms and shows the minimum knowledge required to have locomotion controllers. We then build on this controller and close the loop with RL to improve robustness and performance.

For its application to real robots, DRL requires a solid software base that is trustworthy, allows for fast iterations, and produces reproducible experiments. To address this need, we develop the Stable-Baselines3 (SB3) library, the core of the thesis, which provides reliable and easy-to-use RL implementations. SB3 is complemented by its Jax equivalent SBX for faster experiments and the RL Zoo training framework to ensure reproducibility.

Contents

List of Symbols and Abbreviations

List of Symbols

t	Time
$\mathbf{s}_t$	State (at time t)
$\mathbf{a}_t$	Action (at time t)
r_t	Reward (at time t)
$\pi(\mathbf{s}_t)$	Policy
$\mathcal{N}$	Gaussian distribution
$\mathcal{H}$	Entropy
τ	Torque
q	Joint position
θ	Motor position
k	Stiffness constant
ψ	Phase
φ	Phase shift

List of Abbreviations

BO	Bayesian Optimization
BBO	Black Box Optimization
CPG	Central Pattern Generator
DRL	Deep Reinforcement Learning
HER	Hindsight Experience Replay
DQN	Deep Q-Network
DroQ	Dropout Q-learning
FQI	Fitted Q-Iteration
PD	Proportional Derivative
PPO	Proximal Policy Optimization

RL	Reinforcement Learning
SAC	Soft Actor-Critic
SDE	State-Dependent Exploration
gSDE	Generalized State-Dependent Exploration
SB3	Stable-Baselines3
SBX	Stable-Baselines3 Jax
TQC	Truncated Quantile Critics
ISS	International Space Station

List of Figures

List of Tables

CHAPTER 1

Introduction

The promise of reinforcement learning (RL) in robotics is to enable robots to learn complex tasks through trial and error, without the need for tedious programming or manual control. This is achieved by providing feedback to the agent in the form of a reward, which measures the robot's immediate performance [SB18a].

The goal of RL is to find an optimal behavior, known as an optimal policy, that maximizes a given objective function. By learning from the consequences of its actions, rather than from being explicitly taught, RL becomes a powerful tool for tasks where the optimal behavior is unknown or difficult to specify.

The potential of RL to learn control from interactions without knowledge of the underlying model was first demonstrated in the early 1980s, when RL with discrete actions was used to balance a simulated cart-pole system [BSA83]. The first results on real robots were shown in the 1990s, on a wheeled robot [MC92] and in a peg-in-hole insertion task [GFB94].

As RL algorithms advanced, researchers began to tackle more complex tasks. In the 2000s, RL was successfully applied to quadruped locomotion [KS04], and helicopter control [BS01]. Furthermore, RL was used in conjunction with task-specific parametrization, such as dynamic movement primitives [STS12, SS13], to learn skills such as ball-in-a-cup [SHHR14] and playing table tennis [KPKP14]. These early successes paved the way for the development of more sophisticated RL algorithms and their application to a wide range of robotic tasks.

The advent of deep learning in the 2010s revolutionized the field of RL, leading to the development of deep reinforcement learning (DRL) methods [MKS$^+$15, LHP$^+$16, HZAL18]. These early DRL works demonstrated the potential of learning complex robotic skills using large amounts of data and neural networks [LHP$^+$16]. They also enabled agents to learn from high-dimensional sensory inputs, such as images [LFDA16]. However, learning with more general representations using neural networks came at the cost of sample inefficiency, requiring more data to learn features from scratch. This need for data drove the field towards more simulation and away from real robots [TET12, RVR$^+$17, TFR$^+$17, MBT$^+$18, CB21].

(a) DLR *David* neck

(b) DLR quadruped *bert*

Figure 1.1: The two robots with elastic components that are the main application platforms of this thesis: (a) a tendon-driven robot, (b) a quadruped with series elastic actuators.

1.1 Motivation

In recent years, substantial progress has been made to bring the success of DRL in simulation to the real world [ICT+18, AAC+19, ZQW20]. However, most studies focus on learning from scratch, which is often impractical (requiring thousands of trials) and unsafe when applied directly in the real world [DALM+21]. Training on real robots is rare due to the inherent challenges posed by DRL algorithms, which can lead to dangerous behavior and significant wear and tear [RSS+10, KBP13].

As a result, most work focuses on bridging the simulation-to-reality gap[1] [TFR+17, PAZA18, ABC+20, KFPM21], for instance with the help of accurate simulators, using system identification or learning actuator models [HLD+19, LHW+20]. A common technique is also to heavily randomize the simulation (both robot and environment dynamics) [TFR+17, PAZA18, ABC+20]. Randomization makes the task more difficult for the RL agent, and with no guarantee of successful transfer. This added complexity leads to approaches that are "data-hungry" and rely on massively parallel simulated environments [RHRH22], which is not feasible in a real-world setting.

In contrast, this thesis avoids the sim-to-real gap altogether, with the aim to train directly on the real hardware and guide the learning process with existing expert knowledge. This expert knowledge can take various forms, such as:

- Robot-specific knowledge to design tasks and minimize safety risks, for instance safer exploration
- Prior knowledge for a category of problems, e. g. periodic policies for locomotion tasks
- Controllers that partially solve the task, such as feedforward controllers (e. g. gravity compensation)

By incorporating such expert knowledge, we can enable safe and efficient learning directly on real robots.

In particular, we are interested in applying RL to robots with unique properties: elastic robots (see Fig. 1.1). Elastic robots offer an energy-efficient alternative to traditional rigid

[1] hereafter referred to as sim-to-real

robots, with the ability to store and release energy in their elastic components, providing enhanced robustness through compliance-by-design [DSBG+17]. However, because of their elasticity, they are challenging to model and require the development of novel controllers [KLOAS18]. Machine learning can help overcome the limitations of model-based approaches in this field, making elastic robots an ideal platform to demonstrate the benefits of learning directly on real hardware. As such, they are the primary application focus of this thesis.

In addition to developing new approaches for learning on real robots, we also recognize the importance of reproducibility and reliability in DRL research. The steady escalation of algorithmic complexity in DRL together with poor evaluation practices has led to a reproducibility crisis [RLTK17, HIB+18a, MGR18, PNWW23]. Although a lot of implementations can be found online, too few can be trusted because of the implementation details that are not always presented in the original papers [HIB+18a, HDR+22, HGF+24, PNWW23]. The lack of a common interface for algorithms prevents them from being easily compared or applied to new problems. In this thesis, we aim to fill this gap by providing reliable implementations that share a common interface, together with a training framework that incorporates the best practices for empirical RL. The goal of the software is also to enable DRL on real hardware, allow for fast iterations, and produce reproducible experiments.

1.2 Challenges of Real-Robot Reinforcement Learning

Learning on real hardware closes the gap between simulation and reality, but introduces new challenges that must be addressed.

i. Exploration-Induced Wear and Tear. One major concern is ensuring the safety of the robot during training. Real robots deteriorate over time and even damage themselves if not properly controlled. The exploration phase of RL training is particularly problematic, as it can cause significant wear and tear on the robot. Therefore, it is essential to design safe exploration strategies that minimize mechanical fatigue.

ii. Sample Efficiency. Another challenge is the inability to parallelize training, which is a common practice in simulation-based RL. In simulation, multiple agents can be trained simultaneously, speeding up the learning process. Resetting the robot to an initial state after each trial is also easy in virtual environments. On real hardware, where parallel training is not possible and manual resets are required, it is crucial to be sample efficient.

iii. Real-Time Constraints. In contrast to simulation, the real world cannot be paused or accelerated. This means that the control frequency must be constant, and the time taken to update the policy must be minimized to avoid adding to the existing time delay. Asynchronous updates can partially solve this problem, but fast implementations are required to ensure that the policy improvement step does not affect the control loop. In addition, the learning process can span multiple days, making it vital to be able to reproduce and continue experiments without losing data or performance.

iv. Computational Resource Constraints. Real robots are also subject to sensor noise and limited power resources (e. g. no GPU on embedded platforms). To deploy learning algorithms on real robots, they must be robust to noise and computationally lightweight to run on resource-constrained hardware.

The next section presents the contributions made in this thesis. Each contribution addresses at least one of the challenges discussed above.

1.3 Contributions and Overview

The structure of the thesis is summarized in Fig. 1.2. Table 1.1 presents the main publications on which this dissertation is based, while Table 1.2 lists the publications of secondary contributions.

Chapter 2 provides the background for understanding black-box optimization and reinforcement learning in the context of this thesis. It includes a practical introduction to Deep RL, starting from tabular RL, presenting the main concepts and providing intuitions to understand the algorithms used in this thesis. The next section presents two robots with elastic components and their associated challenges. They are the main application platforms for the contributions presented in this dissertation.

In **Chapter 3**, we develop reliable and easy-to-use RL implementations along with fast variants and a framework that incorporates the best practices for RL experimentation. We contribute software that unifies the interface for using the different algorithms and enables fast and reproducible experiments. These implementations and the training framework are the cornerstones of this thesis. By minimizing implementation errors, they allow learning on real hardware, thus addressing challenges ii. (Sample Efficiency) and iii. (Real-Time Constraints). This chapter is based on [RHG+21].

Chapter 4 addresses the issues of the step-based exploration used by DRL algorithms for continuous control. The contribution is to generalize state-dependent exploration (SDE), leading to the novel algorithm gSDE, which enables successful learning from scratch directly on the real robot without any modifications. This addresses challenge i. (Exploration-Induced Wear and Tear). This chapter is based on [RKS21].

In **Chapter 5**, we contribute a new fault-tolerant, data-driven approach to obtain an accurate pose predictor for an elastic robotic neck. This predictor can be inverted to provide a feedforward controller and enables faster learning of a controller on the hardware, addressing challenges i. (Exploration-Induced Wear and Tear) ii. (Sample Efficiency) and iv. (Computational Resource Constraints). This chapter is based on [RDS21].

In **Chapter 6**, we explore what kind of prior knowledge is helpful for learning locomotion controllers. Addressing challenges ii. (Sample Efficiency) and iv. (Computational Resource Constraints), we contribute a model-free open-loop baseline that can compete with complex DRL algorithms. We then combine open-loop control with a learned feedback controller, thus closing the loop using reinforcement learning. The learned controller allows to exploit the elastic actuators of a quadruped robot and discover new behaviors. This chapter is based on [RSK+22, RSK+24].

Finally, we summarize the thesis in **Chapter 7** and discuss future work.

Table 1.1: Main publications on which the thesis is based.

Reference	Details
[RHG+21] *Journal*	**Raffin, A.**, Hill, A., Gleave, A., Kanervisto, A., Ernestus, M. and Dormann, N., Stable-baselines3: Reliable reinforcement learning implementations. The Journal of Machine Learning Research, 2021.
[RDS21] *Journal*	**Raffin, A.**, Deutschmann, B. and Stulp, F., Fault-tolerant six-DoF pose estimation for tendon-driven continuum mechanisms. Frontiers in Robotics and AI, 2021.
[RKS21] *Conference*	**Raffin, A.**, Kober, J. and Stulp, F., Smooth exploration for robotic reinforcement learning. In Conference on Robot Learning (CoRL), PMLR, 2022.
[RSK+22] *Journal - Submitted*	**Raffin, A.**, Seidel, D., Kober, J., Albu-Schäffer, A., Silvério, J. and Stulp, F., Learning to Exploit Elastic Actuators for Quadruped Locomotion. 2023.
[RSK+24] *Journal*	**Raffin, A.**, Sigaud, O., Kober, J., Albu-Schäffer, A., Silvério, J. and Stulp, F., An Open-Loop Baseline for Reinforcement Learning Locomotion Tasks. RLJ, 2024.

Table 1.2: Publications of secondary contributions.

Reference	Details
[FLO+22] *Preprint*	Franceschetti, M., Lacoux, C., Ohouens, R., **Raffin, A.**, and Sigaud, O., Making Reinforcement Learning Work on Swimmer. 2022.
[HKR+22] *Conference - Submitted*	Huang, S., Kanervisto, A., **Raffin, A.**, Wang, W., Ontañón, S., and Dossa, R. F. J., A2C is a special case of PPO. 2022.
[HDR+22] *Conference*	Huang, S., Dossa, R. F. J., **Raffin, A.**, Kanervisto, A., and Wang, W., The 37 implementation details of proximal policy optimization. The ICLR Blog Track, 2023.
[PQS+23] *Conference*	Padalkar, A., Quere, G., Steinmetz, F., **Raffin, A.**, Nieuwenhuisen, M., Silvério, J., and Stulp, F, Guiding Reinforcement Learning with Shared Control Templates. ICRA, 2023.
[PQR+24] *Journal*	Padalkar, A., Quere, G., **Raffin, A.**., Silvério, J., and Stulp, F., Guiding real-world reinforcement learning for in-contact manipulation tasks with Shared Control Templates. AuRo, 2024.
[VDR+24] *Conference*	Vezzi, F., Ding, J., **Raffin, A.**, Kober, J., and Della Santina, C., Two-Stage Learning of Highly Dynamic Motions with Rigid and Articulated Soft Quadrupeds. ICRA 2024.
[HGF+24] *Conference - Submitted*	Huang, S., Gallouédec, Q., Felten, F., **Raffin, A.**, Dossa, R. F. J., Zhao, Y., ...& Yi, B., Open RL Benchmark: Comprehensive Tracked Experiments for Reinforcement. 2024.
[COR+24] *Conference*	Open X-Embodiment Collaboration, Open X-Embodiment: Robotic Learning Datasets and RT-X. ICRA, 2024.
[SSL+24] *Conference*	Seidel, D., Schmidt, A., Luo, X., **Raffin, A.**, Mayershofer, L., Ehlert, T., Calzolari, D., Hermann, M., Gumpert, T., Loeffl, F., Den Exter, E., Köpken, A., Luz, R., Bauer, A.S., Batti, N., Lay, F.S., Manaparampil, A.N., Albu-Schäffer, A., Leidner, D., Schmaus, P., Krüger, T., and Lii, N.Y., Toward Space Exploration on Legs: ISS-to-Earth Teleoperation Experiments with a Quadruped Robot. IEEE Conference on Telepresence, 2024.

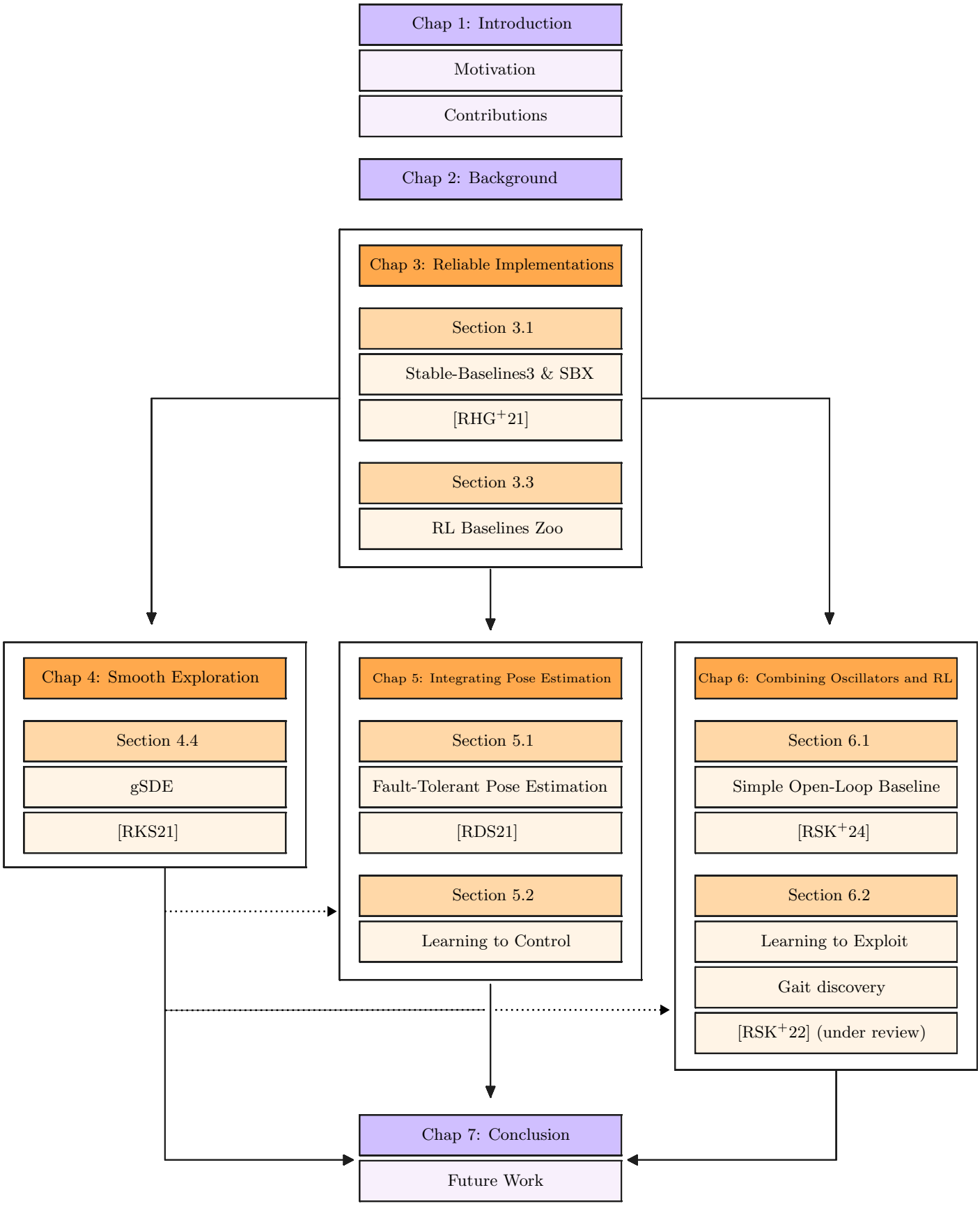


Figure 1.2: Structure of the thesis, highlighting the main themes and contributions of each chapter, and the associated publications. Solid lines show where chapters build on other chapters, and dotted lines show the connection between chapters where results are improved by the use of expert knowledge.

CHAPTER 2

Background

The following sections provide the background for understanding the two main techniques– black-box optimization (BBO) and reinforcement learning (RL)– that are used extensively throughout this thesis. Both techniques are similar in that they attempt to optimize a non-differentiable objective function by updating parameters [SS13], but differ in their assumptions and domain of application.

The next section introduces the two elastic robots that are the main application platforms of this thesis. This chapter concludes with a discussion of the design of real-world RL experiments.

2.1 Black-Box Optimization

In BBO, the goal is to find a set of parameters $\alpha \in \mathbb{R}^n$ that minimizes an objective function $f : \mathbb{R}^n \to \mathbb{R}$,

$$\alpha^* = \arg\min_{\alpha} f(\alpha). \tag{2.1}$$

A BBO algorithm only has access to function evaluations of f. It treats f as a black box and does not make any assumption about it (e. g. f can be non-differentiable).

The objective function f is a measure of how good the chosen parameters are. For example, if we are trying to optimize the parameters α of a robot controller for locomotion, i. e. learning to walk, the objective f can be the average speed achieved by the robot for the given set of parameters. The BBO algorithm does not have access to the intermediate steps of the trial.

One of the simplest BBO algorithms is the 1+1 evolution strategy. The idea is as follows: start with some best-known parameters (random parameters in the first iteration), perturb them randomly, if the objective improves, keep the new parameters, and repeat.

BBO is a very versatile technique that can usually be easily parallelized. Due to its low requirements, it is inefficient for larger problems (number of parameters $\gg 10$), where many function evaluations are needed to find a good solution.

In the cases where the objective function can be decomposed as a sum of immediate rewards, reinforcement learning is better suited to these larger problems.

2.2 A Practical Introduction to Deep Reinforcement Learning

In reinforcement learning, an agent interacts with its environment, usually modeled as a Markov Decision Process[1] (MDP) $(\mathcal{S}, \mathcal{A}, P, r)$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ the action space and $P(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ the transition function. At every step t, the agent performs an action $\mathbf{a}$ in state $\mathbf{s}$ following its policy $\pi : \mathcal{S} \mapsto \mathcal{A}$. It then receives a feedback signal in the next state $\mathbf{s}'$: the reward $r(\mathbf{s}, \mathbf{a})$. The objective of the agent is to maximize the long-term reward. More formally, the goal is to maximize the expectation of the sum of discounted reward, over the trajectories ρ_π generated using its policy π:

$$J = \sum_t \mathop{\mathbb{E}}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[\gamma^t r(\mathbf{s}_t, \mathbf{a}_t) \right] \tag{2.2}$$

where $\gamma \in [0, 1[$ is the discount factor and represents a trade-off between maximizing short-term and long-term rewards. The agent-environment interactions are often broken down into sequences called *episodes*, that end when the agent reaches a terminal state.

In the example of learning to walk, if the goal is to achieve the fastest speed, an immediate reward can be the distance traveled between two timesteps. The state would be the current information about the robot (joint positions, velocities, torques, linear acceleration, ...) and the action would be a desired motor position.

The rest of this chapter is meant to be a practical introduction to (deep) reinforcement learning[2], presenting the main concepts and providing intuitions to understand the algorithms used in this thesis. For a more in-depth and theoretical introduction, we recommend reading [SB18b].

We will start with tabular Q-learning and work our way through Deep Q-Learning (DQN) and other key algorithms to the Soft-Actor Critic (SAC) algorithm.

2.2.1 From Tabular Q-Learning to Deep Q-Learning

In tabular RL, states and actions are discrete, so in this setting it is possible to represent the world as a large table. Each entry corresponds to a state and can be subdivided by the number of possible actions in that state.

Action-Value Function (Q-function)

One key element to solve the discounted RL problem (presented in Section 2.2) is the action-value function, or Q-function, noted Q^π for a given policy π. It is defined as the expected discounted return starting in state s, taking action a, and following policy π:

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau \sim \pi} \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a \right]. \tag{2.3}$$

In other words, the Q-function gives an estimate of how good it is to take the action a in state s while following a policy $\pi(s)$.

The Q-function can be estimated recursively, also known as the Bellman equation:

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{s' \sim P} \left[r(s, a) + \gamma \mathop{\mathbb{E}}_{a' \sim \pi} \left[Q^\pi(s', a') \right] \right]. \tag{2.4}$$

[1] The Markov property states that next state and reward depend only on the current state and action, not on the history of previous states and actions.

[2] For the purposes of this thesis, we focus on value-based methods.

This rewrite allows to build an estimate of the Q-value without having to wait for terminal states. It is the formula used in practice.

By definition of the optimal policy, which selects the actions that **max**imize the expected return, the following optimal Q-function Bellman equation is obtained:

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right]. \tag{2.5}$$

The other way around, if we have the optimal action-value function Q^*, we can retrieve the action taken by the optimal policy π^* using:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s,a). \tag{2.6}$$

Similarly, we can derive a greedy policy from the Q-function:

$$\pi(s) = \arg\max_{a \in A} Q^{\pi}(s,a). \tag{2.7}$$

This policy is implicitly defined: we take the action that maximizes the Q-function. In the tabular case, this action is found by enumerating all possible actions.

For the rest of this chapter, we will drop the π superscript from Q^π so as not to overload the notation (more indices are coming), but unless otherwise noted, Q will always be Q^π.

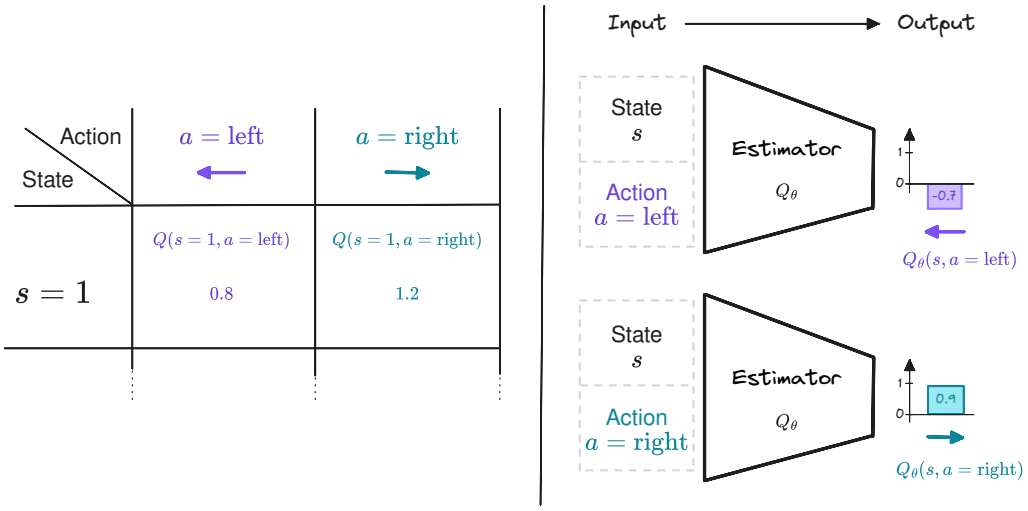


Figure 2.1: Illustration of a Q-Table (left) and Fitted Q-Iteration (FQI) value estimator (right). Compared to the Q-Table, which is limited to discrete states, the FQI value estimator approximates the Q-value for continuous state spaces.

Q-Learning

For discrete states and actions, the Q-learning algorithm can be used to estimate the Q-function of a policy, in this particular case represented as a lookup table (Q-table, shown in Fig. 2.1). The idea is to start with an initial estimate for the first iteration ($n = 0$)[3] and

[3]The initial estimate is usually zero, see [SB18a]

slowly update the estimate over time according to Eqs. (2.4) and (2.5). For each transition tuple (s_t, a_t, r_t, s_{t+1}), we compute the error between the estimation and the target value and update the estimate with a learning rate η:

$$Q^n(s_t, a_t) = Q^{n-1}(s_t, a_t) + \eta \cdot (r_t + \gamma \cdot \max_{a'} Q^{n-1}(s_{t+1}, a') - Q^{n-1}(s_t, a_t)). \tag{2.8}$$

Under the assumptions that all state-action pairs are visited an infinite number of times and that we use a learning rate $\eta \in]0, 1[$, the Q-learning algorithm converges to a fixed point (i. e. $Q^{n+1}(s, a) = Q^n(s, a)$): the optimal action-value function $Q^*(s, a)$.

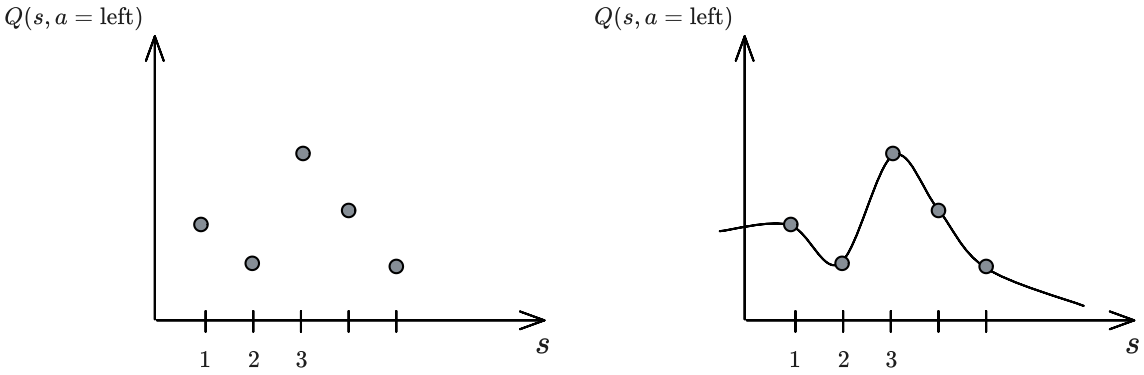


Figure 2.2: Illustration of a Q-function approximated by a lookup table (left, tabular case) and using regression (right, FQI)

The main limitation of Q-learning and its Q-table is that it can only handle discrete states. The size of the table grows with the number of states, which becomes intractable when this number is infinite (continuous states). Moreover, it does not provide any generalization as shown in Fig. 2.2 left: knowing the Q-values for some states does not help to predict the Q-values of unseen states.

Function Approximation and Fitted Q-Iteration (FQI)

A straightforward extension to Q-learning is to estimate the Q-function using function approximation instead of a Q-table, displayed in Figs. 2.1 and 2.2 on the right. In other words, the Q-value estimation problem can be formulated as a regression problem ($f_\theta(X) = Y$):

$$Q^n_\theta(s_t, a_t) = r_t + \gamma \cdot \max_{a' \in \mathcal{A}} (Q^{n-1}_\theta(s_{t+1}, a')) \tag{2.9}$$

$$\mathcal{L}(\theta, X, Y) = \frac{1}{2}(Y - f_\theta(X))^2 \tag{2.10}$$

where $X = (s_t, a_t)$ is the input, $Y = r_t + \gamma \cdot \max_{a' \in \mathcal{A}}(Q^{n-1}_\theta(s_{t+1}, a'))$ is the target, θ are the parameters to be optimized[4], and $\mathcal{L}$ is the loss function. This is similar to what the Q-learning algorithm does in Eq. (2.8).

Since the target Y used to update Q_θ depends on Q_θ itself, we need to iterate. Computing an iterative approximation of the Q-function using regression is the main idea behind

[4]To ease the transition to DQN, we consider only parametric estimators here (i. e. , we exclude kNN for instance)

the Fitted Q-Iteration algorithm (FQI) [EGW05, Rie05], presented in Algorithm 1. This algorithm uses a fixed dataset $\mathcal{D}$ of m transitions (s_t, a_t, r_t, s_{t+1}).

Algorithm 1: Fitted Q-Iteration (FQI)

1. Create the training set based on the previous iteration Q_θ^{n-1} and the m transitions from the dataset $\mathcal{D}$:

$$x_i = (s_t, a_t) \tag{2.11}$$

$$y_i = \begin{cases} r_t + \gamma \cdot \max_{a' \in \mathcal{A}}(Q_\theta^{n-1}(s_{t+1}, a')) & \text{if } s_{t+1} \text{ is non-terminal} \\ r_t & \text{if } s_{t+1} \text{ is terminal} \end{cases} \tag{2.12}$$

2. Fit a model using a regression algorithm to obtain Q_θ^n:

$$f_\theta(x_i) = y_i \tag{2.13}$$

3. Repeat, $n = n + 1$

FQI is a step toward a more practical algorithm, but it still has some limitations:

1. It requires a dataset of transitions $\mathcal{D}$ and does not provide an explicit way to collect new transitions
2. A loop over actions is needed to obtain $\max_{a' \in \mathcal{A}}(Q_\theta^{n-1}(s_{t+1}, a'))$, which is inefficient
3. Because Q_θ^n depends on Q_θ^{n-1}, this leads to instability (the regression target is constantly moving)

Deep Q-Learning (DQN)

The Deep Q-Learning (DQN) algorithm [MKS+13] introduces several components to overcome the limitations of FQI.

First, instead of having a fixed dataset of transitions, DQN uses experience replay [Lin92], also called a replay buffer. The replay buffer, shown in Fig. 2.3, is a first in first out (FIFO) data structure of capacity m, the maximum number of transitions that can be stored. When the buffer is full, old experience is removed. Experience replay provides a compromise between online learning, where transitions are discarded after use, and offline learning, where transitions are stored forever.

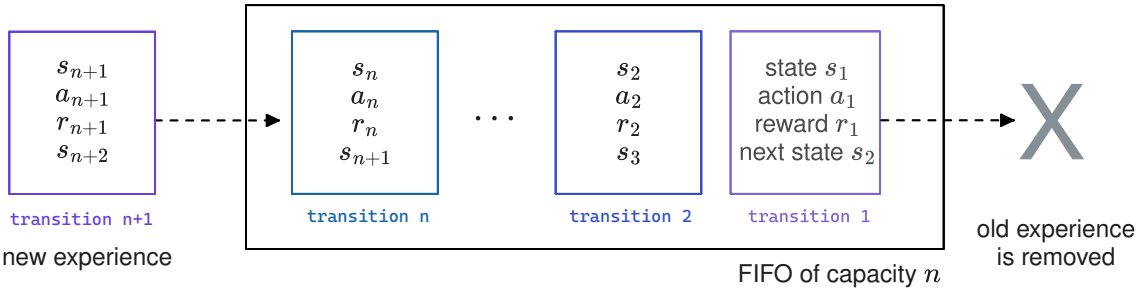


Figure 2.3: DQN replay buffer.

To train its Q-network, DQN creates mini-batches[5] of experience by sampling uniformly from the replay buffer, as illustrated in Fig. 2.4. This breaks the correlation between consecutive samples, allows the agent to learn from diverse experiences (not just the most recent ones), and allows more efficient use of data by reusing past transitions multiple times.

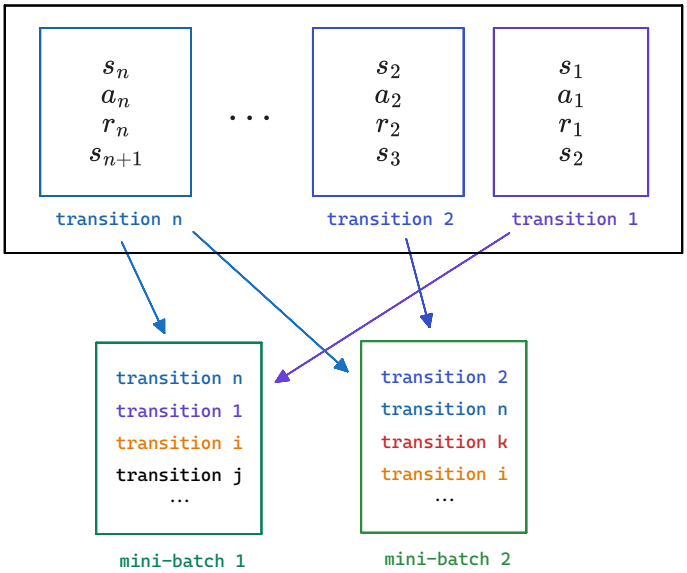


Figure 2.4: DQN replay buffer sampling.

DQN collects samples using an ϵ-greedy strategy (shown in Fig. 2.5): at each step, it chooses a random action with a probability ϵ, or otherwise follows the greedy policy (take the action with the highest Q-value in that state). To balance exploration and exploitation, DQN starts with $\epsilon_{\text{initial}} = 1$ (random policy) and linearly decreases its value until it reaches its final value, usually $\epsilon_{\text{final}} = 0.01$.

Like [Rie05], DQN uses a neural network to approximate the Q-function. However, to avoid the loop over actions of FQI, it outputs all the Q-values for a given state, as shown in Fig. 2.5.

Finally, to stabilize learning, DQN uses an old copy of the Q-network $Q_{\theta_{\text{targ}}}$ to compute the regression target. This second network, the target network, is updated every k steps, so that it slowly follows the online Q-network. Overall, the DQN algorithm, shown in Algorithm 2, is very similar to the FQI algorithm, the main difference being that DQN alternates between collecting new transitions and updating its network.

[5]Rather than doing updates using the entire dataset, it is more practical to perform gradient updates with subsets sampled from the dataset.

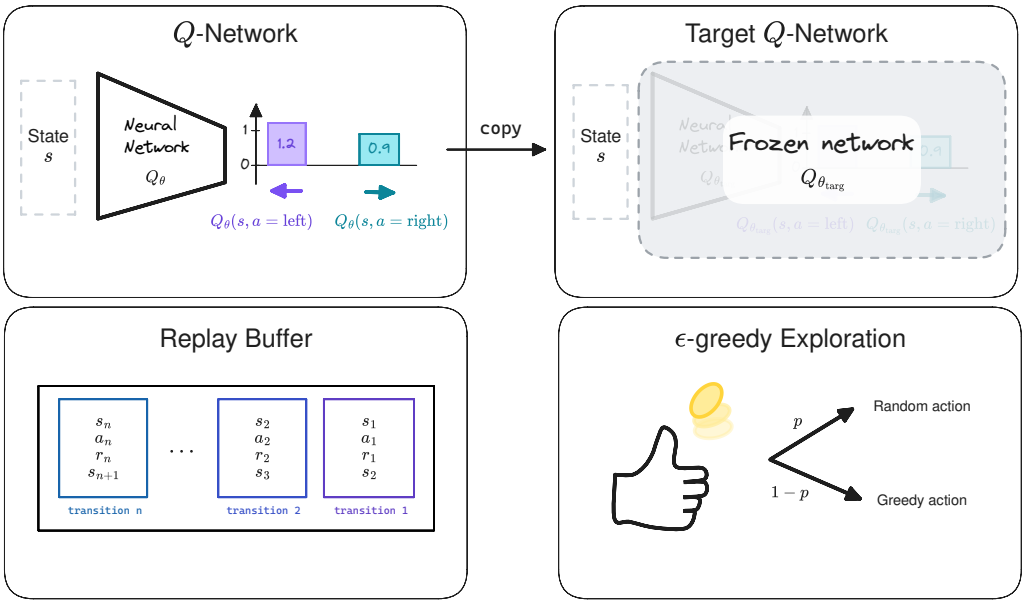


Figure 2.5: Deep Q-Network (DQN) and its main components.

Algorithm 2: Deep Q-Network (DQN)

Initialize replay memory $\mathcal{D}$ of capacity m
Initialize action-value function Q_θ with random weights θ
Set target parameters equal to online parameters $\theta_{\text{targ}} \leftarrow \theta$
Initialize the environment and retrieve s_1 (initial state)
for $t = 1, T$ **do**
  With probability ϵ, select a random action a_t,
  otherwise use the greedy policy $a_t = \arg\max_{a \in \mathcal{A}} Q_\theta(s_t, a)$
  Execute action a_t in the environment
  Observe reward r_t and next state s_{t+1}
  Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  Sample random mini-batch of transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{D}$
  Set $y_i = \begin{cases} r_i + \gamma \max_{a'} Q_{\theta_{\text{targ}}}(s_{i+1}, a') & \text{for non-terminal } s_{i+1} \\ r_i & \text{for terminal } s_{i+1} \end{cases}$
  Perform a gradient descent step on $(y_i - Q_\theta(s_i, a_i))^2$
  Every k steps, set target parameters equal to online parameters $\theta_{\text{targ}} \leftarrow \theta$
end for

The key components of DQN (Q-network, target network, replay buffer) are at the core of Deep RL algorithms for continuous control used on real robots such as Soft Actor-Critic (SAC). We introduce these algorithms in the next section.

2.2.2 From Deep Q-Learning to Soft Actor-Critic and Beyond

While FQI and DQN algorithms can handle continuous state spaces, they are still limited to discrete action spaces. Indeed, all possible actions ($a \in \mathcal{A}$) must be enumerated to compute $\max_{a' \in \mathcal{A}} Q_\theta^{n-1}(s_{t+1}, a')$ or $\arg\max_{a \in \mathcal{A}} Q(s, a)$ Eq. (2.9) and Algorithm 2, used to update the Q-value estimate and select the action according to the greedy policy.

One solution to enable Q-learning in continuous action space is to parametrize the Q-function so that its maximum can be easily and analytically determined. This is what the Normalized Advantaged Function (NAF) [GLSL16] does by restricting the Q-function to a function quadratic in a.

Extending DQN to Continuous Actions: Deep Deterministic Policy Gradient (DDPG)

Another possibility is to train a second network $\pi_\phi(s)$ to maximize the learned Q-function[6]. In other words, π_ϕ is the actor network with parameters ϕ and outputs the action that leads to the highest return according to the Q-function:

$$\max_{a \in A} Q_\theta(s, a) \approx Q_\theta(s, \pi_\phi(s)). \tag{2.14}$$

This idea, developed by the Deep Deterministic Policy Gradient (DDPG) algorithm [DCH+16], provides an explicit deterministic policy π_ϕ for continuous actions. Since Q_θ and π_ϕ are both differentiable, the actor network π_ϕ is directly trained to maximize $Q_\theta(s, \pi_\phi(s))$ using samples from the replay buffer $\mathcal{D}$ as illustrated in Fig. 2.6:

$$\mathcal{L}_\pi(\phi, \mathcal{D}) = \max_\phi \mathop{\mathbb{E}}_{s \sim \mathcal{D}} \left[Q_\theta(s, \pi_\phi(s)) \right]. \tag{2.15}$$

For the update of the Q-function Q_θ, DDPG uses the same regression target as DQN.

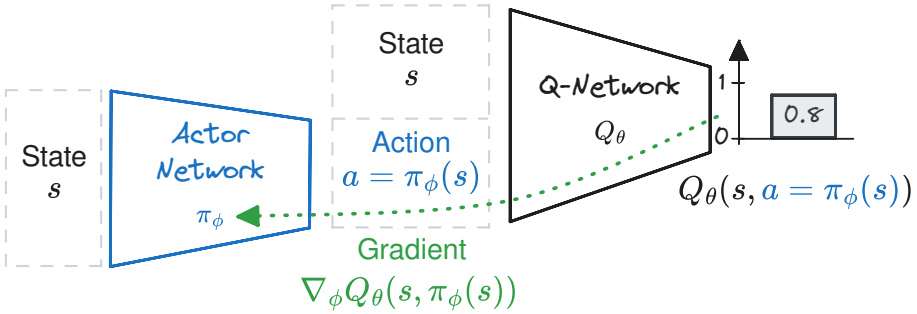


Figure 2.6: DDPG update of the actor network. The gradient computed using the loss of Eq. (2.15) is backpropagated through the Q-network to update the actor network so that it maximizes the Q-function.

DDPG extends DQN to continuous actions but has some practical limitations. π_ϕ tends to exploit regions of the state space where the Q-function overestimates the Q-value [FvHM18], as shown in Fig. 2.7. These regions are usually those that are not well covered by samples from the buffer $\mathcal{D}$. Because of this interaction between the actor and critic networks, DDPG is also often unstable in practice (divergent behavior).

Twin Delayed DDPG (TD3) and Soft Actor-Critic (SAC)

To overcome the limitations of DDPG, Twin Delayed DDPG (TD3) [FvHM18] employs three key techniques:

1. Twin Q-networks: TD3 uses two separate Q-networks and selects the minimum Q-value estimate from the two networks. This helps to reduce overestimation bias in the Q-value estimates.

[6] A third option is to sample the Q-value, as explored by QT-Opt [KIP+18].

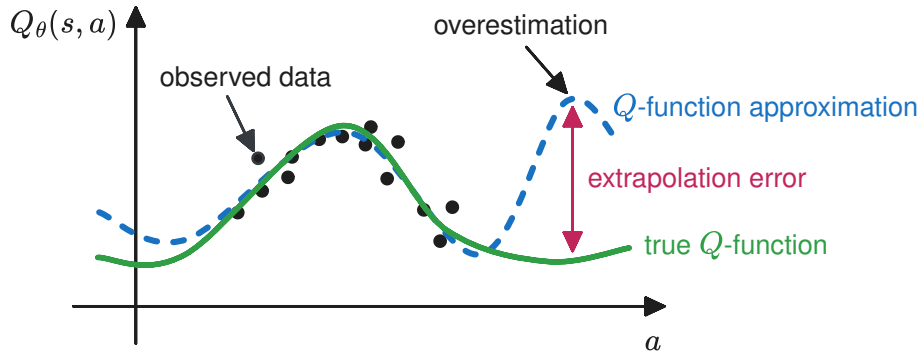


Figure 2.7: Illustration of the overestimation and extrapolation error when approximating the Q-function. In regions where there is training data (black dots), the approximation matches the true Q-function. However, outside the training data support, there may be extrapolation error (in red) and overestimation that the actor network can exploit.

2. Delayed policy updates: TD3 updates the policy network less frequently than the Q-networks, allowing the policy network to converge before being updated.

3. Target action noise: TD3 adds noise to the target action during the Q-network update step. This makes it harder for the actor to exploit the learned Q-function.

Since TD3 learns a deterministic actor network π_ϕ, it relies on external noise during the exploration phase. A common approach is to use a step-based Gaussian noise[7]:

$$\mathbf{a}_t = \pi_\phi(\mathbf{s}_t) + \epsilon_t, \qquad \epsilon_t \sim \mathcal{N}(0, \sigma^2). \tag{2.16}$$

While the standard deviation σ is usually kept constant, it is a critical hyperparameter that gives a compromise between exploration and exploitation [PHD+17, Raf20, HGF+24].

To better balance exploration and exploitation, Soft Actor-Critic (SAC) [HZAL18], successor of Soft Q-Learning (SQL) [HTAL17], optimizes the maximum-entropy objective, which is slightly different from the classical RL objective Eq. (2.2):

$$J(\pi) = \sum_t \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[\gamma^t r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\cdot|\mathbf{s}_t)) \right] \tag{2.17}$$

where $\mathcal{H}$ is the policy entropy and α is the entropy temperature, allowing to have a trade-off between the two objectives. This objective encourages exploration by maximizing the entropy of the policy while still solving the task by maximizing the expected return (classic RL objective).

SAC learns a stochastic policy using a squashed Gaussian distribution, and incorporates the clipped double Q-learning trick from TD3. In its latest iteration [HZH+18], SAC automatically adjusts the entropy coefficient α, eliminating the need to tune this crucial hyperparameter.

[7] Chapter 4 will present more details about exploration for continuous action spaces

Algorithm 3: Soft Actor-Critic (SAC)

Initialize replay memory $\mathcal{D}$ of capacity m
Initialize Q-networks Q_{θ_1}, Q_{θ_2} and actor network π_ϕ with random weights
Set target parameters equal to online parameters $\theta_{\text{targ},1} \leftarrow \theta_1$, $\theta_{\text{targ},2} \leftarrow \theta_2$
Initialize the environment and retrieve s_1 (initial state)
for $t = 1, T$ **do**
Select action $a_t \sim \pi_\phi(s_t)$
Execute action a_t in the environment
Observe reward r_t and next state s_{t+1}
Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
Sample random mini-batch of transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{D}$
Compute targets for the Q functions:

$$y_i = r_i + \gamma(1 - d_i)\left(\min_{j=1,2} Q_{\theta_{\text{targ},j}}(s_{i+1}, a') - \alpha \log \pi_\phi(s_{i+1})\right), \quad a' \sim \pi_\phi(s_{i+1})$$

where $d_i = 1$ if s_{i+1} is terminal, $d_i = 0$ otherwise

Update Q-functions by one step of gradient descent using

$$\nabla_{\theta_j}\left(y_i - Q_{\theta_j}(s_i, a_i)\right)^2 \qquad \text{for } j = 1, 2$$

Update policy by one step of gradient ascent using

$$\nabla_\phi\Big(\min_{j=1,2} Q_{\theta_j}(s_i, a_\phi) - \alpha \log \pi_\phi(s_i)\Big), \qquad a_\phi \sim \pi_\phi(s_i)$$

Update target networks with

$$\theta_{\text{targ},j} \leftarrow \rho\theta_{\text{targ},j} + (1 - \rho)\theta_j \qquad \text{for } j = 1, 2$$

end for

In summary, as shown in Algorithm 3, SAC combines several key elements from the algorithms presented in this chapter. It uses the update rule from FQI and adopts the Q-network, target network and replay buffer from DQN to learn the Q-function. SAC also incorporates techniques from DDPG to handle continuous actions, uses the clipped double Q-learning trick from TD3 to reduce overestimation bias, and optimizes the maximum entropy objective with a stochastic policy to balance exploration and exploitation.

SAC and its variants are the algorithms used in this thesis to train RL agents directly on real robots.

Beyond SAC: TQC, REDQ, DroQ, ...

Several extensions of SAC have been proposed, in particular to improve the sample efficiency. One notable example is Truncated Quantile Critics (TQC) [KSGV20] which builds upon SAC by incorporating distributional RL [BDM17]. In distributional RL, the Q-function estimates the distribution of returns instead of just the expected return. Fig. 2.8 illustrates the benefits of learning the distribution of returns rather than only the expected value in an example.

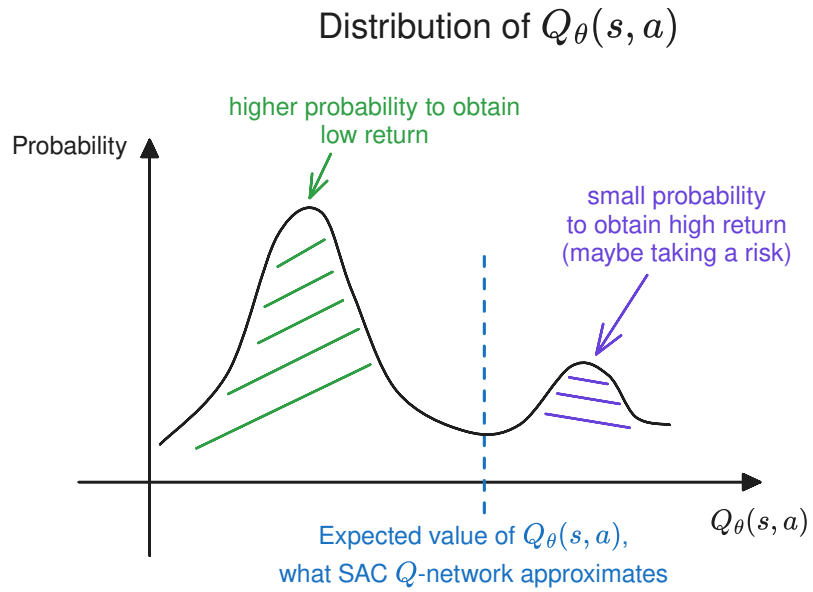


Figure 2.8: An example where learning the distribution of returns (distributional RL) instead of the expected value (classic RL) can be useful. We plot the distribution of returns for a given state-action pair (s, a). In this case, there is a bimodal distribution. Learning the expected value of it instead of the distribution itself is harder and does not allow to measure the risk of taking a particular action.

A key idea to improve sample efficiency is to perform multiple gradient updates for each data collection step. However, simply increasing the update-to-data (UTD) ratio may not lead to better performance due to the overestimation bias. To address this issue, the algorithms REDQ [CWZR21] and DroQ [HIH+22] rely on ensembling techniques (explicit for REDQ, implicit for DroQ with dropout). Finally, a new algorithm, CrossQ [BPB+24], takes a different approach by removing the target network and using batch normalization to stabilize learning.

2.3 Elastic Robots System Description: Hardware and Challenges

Robotic systems with deliberately introduced elasticity are a promising alternative to their rigid counterparts [MKH21, BLA+23]. In addition to the passive compliance, which refers to the ability to yield to external forces without breaking, the elastic elements provide energy storing and improved efficiency if properly used. Designing controllers that take full advantage of the intrinsic dynamics of elastically actuated robots remains a challenge, as it requires a considerable amount of time and system-specific knowledge [KLOAS18]. In this thesis, we address the problem of obtaining efficient controllers for robots with elastic components by formulating controller design as a learning problem.

This section introduces two robotic platforms with elastic components and are used extensively in this thesis: the *David* elastic neck and the *bert* quadruped robot. Further, we present the unique challenges posed by the platforms in controlling them effectively.

2.3.1 David Elastic Neck

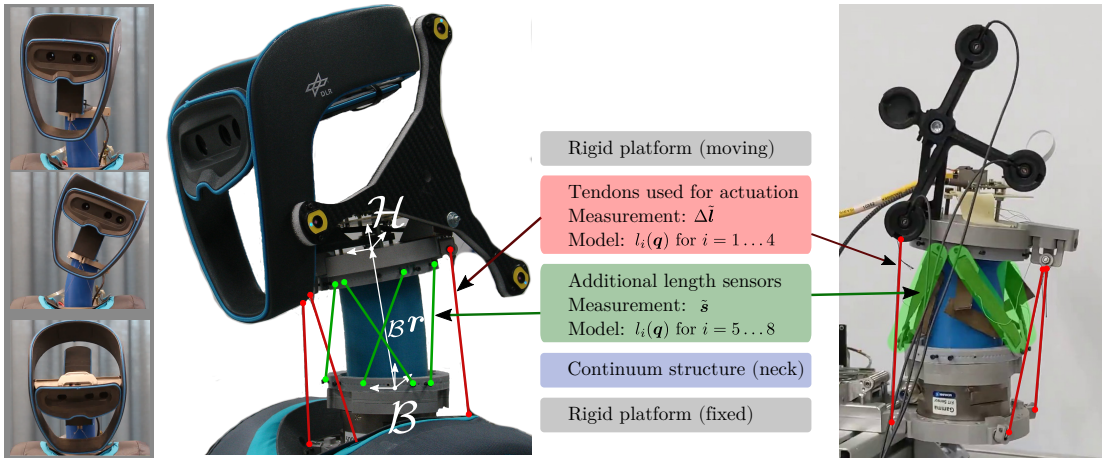


Figure 2.9: Neck and head of the humanoid robot DLR *David*. Left: Demonstration of the structural flexibility and range of motion of the neck joint. Middle: Detailed schematic of the experimental system. Right: Experimental setup showing the LEDs used by the tracking system (in black) and the additional length sensors (highlighted in green).

The *David* elastic neck [RDF16] is a tendon-driven continuum mechanism, as shown in Fig. 2.9. It consists of a fixed lower platform, a moving upper platform and a continuum structure in between. The inertial frame of reference, denoted by $\mathcal{B}$, is attached to the base of the neck. The position of the upper platform is described by the origin of the frame $\mathcal{H}$, expressed in $\mathcal{B}$, and is denoted by ${}_{\mathcal{B}}\boldsymbol{r} = (x,\, y,\, z)^T$. The orientation of frame $\mathcal{H}$, expressed in $\mathcal{B}$, is represented by the three Euler angles, denoted as $\boldsymbol{\theta} = (\theta_x,\, \theta_y,\, \theta_z)^T$. Since the workspace of the system does not exceed $\pm 90°$ in any direction, no singular configurations of the Euler angles appear. The pose is summarized in the vector $\boldsymbol{q} \in \mathbb{R}^6$, which includes the position and orientation of the platform:

$$\boldsymbol{q} = (x,\, y,\, z,\, \theta_x,\, \theta_y,\, \theta_z)^T\,. \tag{2.18}$$

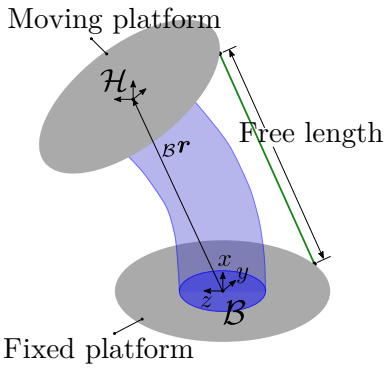


Figure 2.10: Schematic of the tendon-driven continuum mechanism, showing the coordinate frames.

The tendons are connected to the upper platform and run along the continuum to the lower platform without touching, as shown in red in Fig. 2.9. The actuators are located in the lower platform and allow deformation of the continuum by applying tension to the tendons. The position displacement $\Delta\tilde{\boldsymbol{l}} \in \mathbb{R}^4$ allows the free length of each tendon to be measured, assuming a known initial length $\boldsymbol{l}_{t,0} \in \mathbb{R}^4$,

$$\boldsymbol{l}_t = \boldsymbol{l}_{t,0} - \Delta\tilde{\boldsymbol{l}}. \tag{2.19}$$

Additionally, four length sensors are placed on the system, shown in green in Fig. 2.9, which provide sensor values $\tilde{\boldsymbol{s}} \in \mathbb{R}^4$ that are linearly proportional to their length:

$$\boldsymbol{l}_s = \boldsymbol{K}_s\tilde{\boldsymbol{s}}, \tag{2.20}$$

with the constant calibration matrix $\boldsymbol{K}_s \in \mathbb{R}^{4\times4}$.

In summary, the *David* elastic neck is a tendon-driven continuum mechanism that uses tendons and length sensors to control the motion of the upper platform. The pose of the upper platform is described by the position and orientation of the platform, and the sensor information $\tilde{\boldsymbol{l}} = (\boldsymbol{l}_t, \boldsymbol{l}_s) \in \mathbb{R}^8$ is used to measure the length of the tendons and additional length sensors.

The *David* elastic neck presents several challenges because of the deformation of its continuum structure, which cannot be measured directly. To effectively control the robot, it is crucial to accurately measure the tendon lengths and determine the position and orientation of the upper platform. In addition, achieving precise control is difficult and requires a complex strategy due to the non-linear relationship between the motion of the tendons and the upper platform.

2.3.2 DLR Quadruped bert

Figure 2.11: Compliantly actuated DLR quadruped *bert* jumping in place.

The DLR quadruped *bert* [SHG+20] (shown in Fig. 2.11) is a cat-sized quadruped robot with series elastic actuators (SEA). The legs are attached to the trunk through the hip axes, while the motors are located inside the trunk and connected by belt drives. Each leg consists of a hip and a knee joint and is composed of two segments of equal length of approximately 8cm. The total mass of the robot is about 3.1kg. As depicted in Fig. 2.12, motors are connected to the links via a linear torsional spring with constant stiffness $k \approx 2.75\text{Nm/rad}$. We use the deflection of the spring to estimate the external torque applied to each joint:

$$\tau_i = k(\theta_i - q_i) \tag{2.21}$$

where θ is the motor position before the spring and q the joint position.

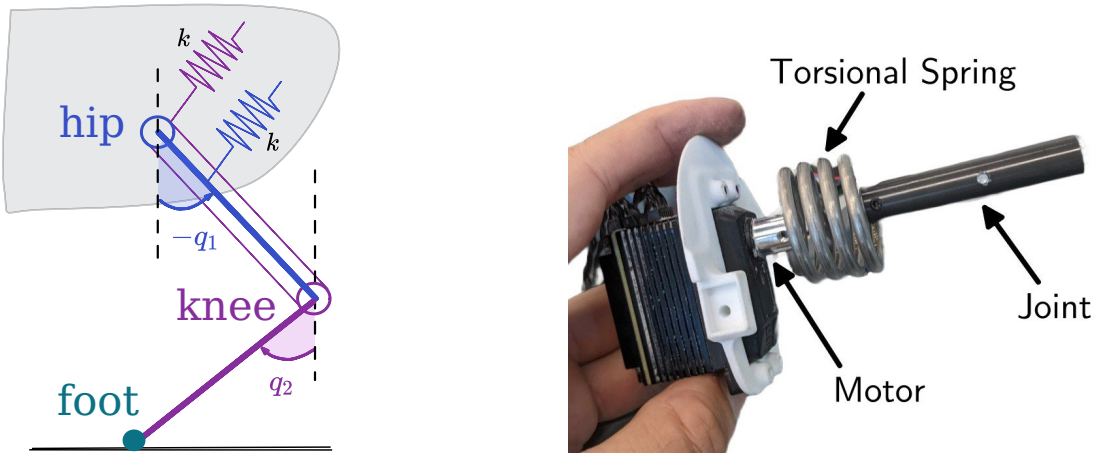


Figure 2.12: Model of the compliantly actuated leg (left) and a serial elastic actuator (right).

The springs on *bert* are extremely soft compared to its weight and size, i. e. 2.75Nm/rad, compared to for instance StarlETH [HGB+12] with 20cm link length, total weight of 23kg and k=70Nm/rad. This means that, for example, when standing in a default position with joints at 30 deg, *bert* can be pushed into joint limits without motor movement, while StarlETH can be pushed into its springs only minimally. This unique characteristic makes *bert* well-suited for using its springs for energy storage. Forces required to load the springs are quite low and occur during normal gait motions. When

released at the right time, the stored energy can be converted into kinetic energy (joint velocity), potentially improving its performance.

Finally, *bert* was built with low cost in mind. The robot's chassis is entirely 3D printed, and it uses off-the-shelf servo motors with custom electronics. Due to their low torque output, these motors have a very high gear ratio (1:325), which results in a generally poor efficiency of the entire drivetrain. The combination of soft springs and low-cost motors presents both a challenge and an opportunity: the robot's elasticity must be exploited to make it move. Just like pushing a child's swing at the wrong frequency, controlling *bert* against its natural dynamics is inefficient and ineffective.

2.4 Designing Real-World Reinforcement Learning Experiments

As discussed in the introduction, while RL has shown remarkable success in simulated environments, translating these results to the real world presents unique challenges. Real-world experiments require careful consideration of safety, efficiency, and reproducibility, among other factors. In this section, we will explore key aspects of designing real-world RL experiments, from task design and algorithm selection to safety considerations. By exploring best practices and strategies for overcoming common challenges, we aim to provide a starting point for researchers and practitioners seeking to use RL on real robots.

2.4.1 Starting Simple: Gall's Law

When designing RL experiments, a key principle is to always start simple. This applies to various aspects such as selecting the problem to solve and choosing the reward function. Complexity should be introduced only after the simple problem is well understood and solved. This principle is summarized by Gall's law [Gal77]:

> A complex system that works is invariably found to have evolved from a simple system that worked. The parallel proposition also appears to be true: a complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a simple system.

Although it is tempting to begin with the interesting and complex problem we want to solve, it is important to start with a rudimentary version of it. By gradually increasing the complexity, we have a better understanding of the problem, what works and what does not, and the implications of each design decision. Having an elementary version of the task allows for faster iterations: instead of waiting hours to know if a single change has an effect, we can try different variants in minutes.

2.4.2 Task Design

Task design is the most critical aspect of an RL experiment, as it determines how the agent perceives and interacts with its environment. This involves defining the observation space, action space, reward function, and termination conditions.

It is important to normalize these quantities, ensure the observation space contains sufficient information to solve the task, and avoid violating the Markov assumption. The aim is to create a well-defined environment for the RL agent to learn and perform efficiently.

Observation Space

The observation space specifies the information available to the agent at each time step [TTK+23]. It is essential to ensure that the observation space contains sufficient information for the agent to solve the task while avoiding the inclusion of irrelevant details that could slow down learning. For example, if the agent is learning to balance a pole on a cart, the observation space should include the cart's position, the pole's angle, and their respective velocities.

To maintain the Markov property, the observation space should provide all necessary information for the agent to make decisions based solely on the current state [SB18a]. Violating the Markov assumption can lead to suboptimal performance [PTLK17]. For instance, in the case of the cart-pole task, removing the velocity from the observation

makes it impossible to balance the pole without memory (e. g. using a recurrent neural network).

Since Deep RL algorithms are optimized using gradient descent, normalizing the observation space can improve learning stability and speed [HDR+22, HGF+24]. This involves scaling the observations to a range between -1 and 1, which can be achieved using a running average if the limits are not known in advance.

Action Space

The action space defines the set of actions available to the agent in each state to interact with the environment. In robotics, continuous action spaces are often more suitable than discrete ones, as they allow for finer control granularity. However, there is a trade-off between the complexity of the action space and the final performance.

A larger action space provides the agent with more flexibility, potentially leading to better performance at the cost of slower learning. Conversely, a smaller action space can accelerate learning but may result in suboptimal solutions. For example, when learning to race with a car, restricting the agent's steering range reduces the search space but prevents it from going faster in sharp turns [Raf21].

When using continuous action spaces, it is recommended to normalize actions to the range of -1 to 1 [HDR+22]. This recommendation comes from the fact that many RL algorithms for continuous action spaces rely on a Gaussian (normal) distribution to represent the policy. This distribution is commonly initialized with a mean of zero and a standard deviation of one. If the boundaries of the action space are not normalized, the sampling of actions from this distribution can lead to inefficient learning or undesirable behavior[8].

For example, if the action limits are too large (e. g. -1000 to 1000), the sampled actions will be clustered near zero, far from the actual action space boundaries, limiting exploration and the agent's ability to use the full range of actions. In the opposite case, if the limits are too small (e. g. -0.02 to 0.02), the sampled actions will often exceed these limits, leading to saturation (actions outside the defined range are usually clipped). This clipping of actions can make debugging difficult and hinder the learning process.

Normalizing the action space to -1 and 1 matches the initial standard deviation of the Gaussian distribution, ensuring that the sampled actions are well distributed within the action space. In addition, normalization allows for flexibility in scenarios where action limits may change during training, as the normalized actions can be rescaled to match the dynamic limits of the environment [PQR+24].

Reward Function

The reward function is a fundamental component of an RL experiment as it defines the objective the agent is trying to achieve. A well-crafted reward function will guide the agent toward learning the desired behavior, while a poorly designed one can lead to unintended or undesirable strategies. An example of unintended consequence is called reward hacking, where the agent exploits loopholes in the reward function to maximize its reward without achieving the intended goal.

Following Gall's law, it is recommended to start with a simple, shaped reward and gradually increase complexity over time. Reward shaping [NHR99] is a technique that can facilitate learning by providing the agent with intermediate rewards that guide it towards

[8] See `https://github.com/DLR-RM/stable-baselines3/issues/1450` and related issues.

the ultimate goal. For instance, in a maze navigation task, the agent can receive small rewards for moving closer to the exit, helping it learn the optimal path more efficiently.

When designing the reward function, it is also useful to distinguish between primary and secondary rewards. The primary reward represents the main goal of the task, while secondary rewards encourage other desirable behaviors. For example, in a robot locomotion task, the primary reward could be the distance traveled, and the secondary reward could be minimizing energy consumption. To simplify the tuning of weights between different reward components, it is recommended to normalize the reward terms. When reward terms have the same magnitude, it is easier to assign priority to each of them.

Termination Conditions

Termination conditions determine when an episode should end. Common termination conditions include early stopping or time limits [TTK+23].

Early stopping can improve learning speed by preventing the agent from spending time in irrelevant parts of the environment. For example, in a robot locomotion task, the episode could be ended if the agent falls. However, care must be taken with early stopping, as it can lead to reward hacking if the agent learns to deliberately fail early to maximize its reward.

Choosing appropriate termination conditions is critical because they can significantly influence the agent's learning. For example, if the agent is penalized at every time step, but the episode only ends when the goal is reached or when the robot fails (early stopping), the agent might learn to fail quickly to maximize its reward. Another case of failure in this scenario is that the agent learns to get to the goal as fast as possible, but may overshoot at deployment time (if the episode ends as soon as the goal is reached).

Timeouts (or time limits) are commonly used to ensure episodes do not run indefinitely [TTK+23]. However, if not handled correctly, timeouts can violate the Markov assumption. One approach to address this issue is to include the remaining time in the observation space [PTLK17]. Alternatively, the algorithm can be modified to use an infinite horizon variant.

Handling Truncations: Infinite Horizon Tasks

In infinite horizon tasks, such as robot locomotion, distinguishing between truncation and termination is essential. Truncation occurs when an episode ends due to external factors like time limits or reaching the boundaries of the task, while termination happens due to the agent's actions, such as falling. We illustrate these two concepts in Fig. 2.13.

Proper handling of truncation is crucial to avoid penalizing the robot for events it cannot control. This involves treating truncated episodes as if they had continued indefinitely. The distinction between truncation and termination can significantly impact the learning process [PTLK17, RKS21, HGF+24].

An example that shows the effect of handling truncations is the derivation of the value function in a simple case. We consider a task with an artificial time limit of four (i. e. there are four steps per episode). At each step, the agent receives a reward of one ($\forall t, \quad r_t = 1$). With a discount factor of $\gamma = 0.98$, according to the definition of the value function (see Section 2.2), we get two different values for the initial state, depending on whether we take the time limit into account or not. If we treat the timeout as a normal termination:

$$
\begin{aligned}
V_\pi(s_0) &= \sum_{t=0}^{3}[\gamma^t r_t] \\
&= 1 + 1 \cdot 0.98 + 0.98^2 + 0.98^3 \approx 3.9.
\end{aligned}
$$

If we handle the truncation properly, treating the problem as an infinite horizon problem, we obtain:

$$
\begin{aligned}
V_\pi(s_0) &= \sum_{t=0}^{\infty}[\gamma^t r_t] \\
&= \sum_{t=0}^{\infty}[\gamma^t] \quad (\forall t, \quad r_t = 1) \\
&= \frac{1}{1-\gamma} \approx 50 \quad \text{(geometric series)}.
\end{aligned}
$$

This simple example illustrates the impact of appropriate handling of truncation on the value function. It may have similar effects in more complex scenarios.

2.4.3 Algorithm Selection

Choosing the appropriate algorithm for an RL experiment depends on several factors, such as the type of action space (continuous or discrete), the desired balance between learning speed (training time) and sample efficiency, and the feasibility of parallelization.

For scenarios that prioritize fast learning and the ability to parallelize, on-policy algorithms like Proximal Policy Optimization (PPO) [SWD+17] or Advantage Actor-Critic (A2C) [MBM+16, HKR+22] are well-suited. When dealing with discrete action spaces and the need for high sample efficiency, Deep Q-Network (DQN) [MKS+13] and its variants, including Rainbow [HMVH+18] and Quantile Regression DQN (QR-DQN) [DRBM18], are particularly effective. In continuous action spaces where sample efficiency is crucial, off-policy algorithms from the Soft Actor-Critic (SAC) family, such as Truncated Quantile Critics (TQC) and Dropout Q-Functions (DroQ), are recommended (see Section 2.2.2). Evolution strategies, on the other hand, can be effective when extensive parallelization is possible, enabling large-scale, asynchronous experiments [SHC+17].

2.4.4 Safety Layers

Ensuring the safety of real-world RL experiments is of utmost importance, especially when using model-free algorithms that are inherently unsafe (as we will see in Chapter 4). One approach to mitigate risks is to carefully design the action space to limit dangerous actions. For example, controlling bounded tendon forces instead of motor positions in a robotic neck can prevent tendon damage.

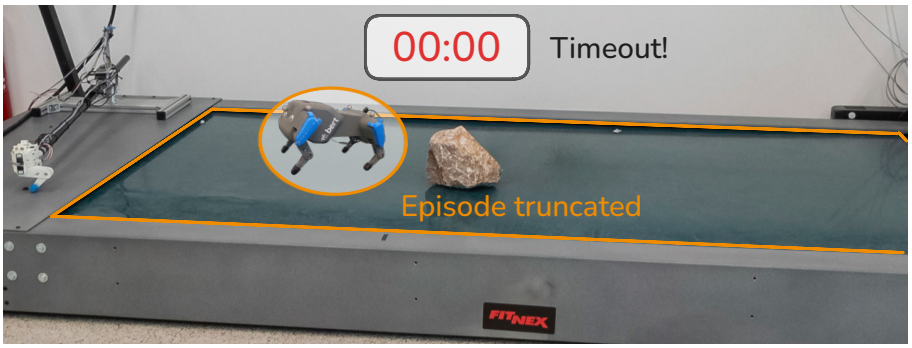
Another approach to safety is implementing hard constraints or safety layers that override the agent's actions if they violate safety limits [QHI+20, LTAP22, PQS+23, PQR+24]. For example, a safety layer could prevent a robotic arm from knocking over objects or spilling liquids. These constraints ensure that the agent operates within safe boundaries, even during exploration.

Incorporating prior knowledge can also enhance safety and efficiency in RL experiments. By leveraging existing controllers or domain-specific knowledge, the search space for the

RL agent can be reduced, guiding it towards safer and more efficient solutions. This strategy will be explored notably in Chapter 5 and Chapter 6.

(a) Experimental setup of a quadruped robot on a treadmill.

(b) The episode terminates because the robot falls (early stopping)



(c) The episode ends because the time limit has been reached (truncation).

(d) The episode ends because the robot reached the limits of the tracking device (truncation).

Figure 2.13: Illustration of the difference between termination and truncation in a locomotion task setup (a). The robot must learn to walk forward as fast as possible without falling (termination, as in (b)). If the robot reaches the limit of the tracking device or the time limit (here a maximum of 15s per episode), the episode ends, but it is considered a truncation (as seen in (c) and (d)).

CHAPTER 3

Reliable Software for Reproducible RL Research

Results in RL are often difficult to reproduce [HIB+18b] and require a careful development process so that no important details are overlooked [HDR+22]. Trusted implementations are especially needed when learning directly on a real hardware, where interaction with the environment is costly–as discussed in challenges ii. (Sample Efficiency) and iii. (Real-Time Constraints)– and potential bugs can affect the performance. Furthermore, it is important to have fast and reproducible experiments, keep track of changes, and be able to isolate the key factors leading to success or failure.

To address these challenges, this chapter presents reliable implementations of DRL algorithms along with fast variants and a training framework for reproducible experiments. This software, developed during the course of this thesis, forms its core: every subsequent chapter rely on it.

3.1 Stable-Baselines3 (SB3): Reliable RL Implementations

A major challenge in RL is that small implementation details can have a substantial effect on performance – often greater than the difference between algorithms [EIS+20, HDR+22]. It is particularly important that implementations used as experimental *baselines* are reliable; otherwise, novel algorithms compared to weak baselines lead to inflated estimates of performance improvements.

To address this challenge, we developed Stable-Baselines3 (SB3), an open-source framework that implements ten popular model-free DRL algorithms (see Section 3.1.2) using the Gym interface [TTK+23]. To ensure high-quality implementations, we follow software engineering best practices, including automated tests, static type checking, and code reviews. Each algorithm is benchmarked on common environments [RKS21] and compared to prior implementations. Our comprehensive test suite covers 96% of the code, and our active user base[1] scrutinizing changes, helps minimize implementation errors.

Building on our experience with Stable-Baselines2 (SB2) [HRE+18], a previous implementation forked from OpenAI Baselines [DHK+17], we have completely rewritten the

[1] At the time of writing, SB3 has 5M downloads on PyPi, has 9000+ stars, 1400+ closed issues and 500+ merged pull requests on GitHub, and the accompanying paper [RHG+21] has 2500+ citations.

codebase in PyTorch [PGM+19]. SB3 maintains a similar API to SB2, allowing for a seamless upgrade pathway[2].

```python
import gymnasium as gym
from stable_baselines3 import SAC

# Train an agent using Soft Actor-Critic on Pendulum-v1
env = gym.make("Pendulum-v1")
model = SAC("MlpPolicy", env).learn(total_timesteps=20_000)
# Save the model
model.save("sac_pendulum")
# Load the trained model
model = SAC.load("sac_pendulum")
# Start a new episode
obs, _ = env.reset()
# What action to take in state `obs`?
action, _ = model.predict(obs, deterministic=True)
```

Figure 3.1: Using Stable-Baselines3 to train, save, load, and infer an action from a policy.

3.1.1 Design Principles

Our main objective is to provide a user-friendly and reliable RL library. To keep SB3 simple to use and maintain, we focus on model-free and single-agent RL algorithms, and leverage external projects for imitation [WTGE20] and offline learning [Sen20]. We prioritize maintaining *stable* implementations rather than adding new features or algorithms, and avoid making breaking changes. We provide a consistent, clean and fully documented API, inspired by the `scikit-learn` API [PVG+11]. Our code is designed to be readable and simple, using object-oriented programming to minimize duplication, making it easy for users to modify.

3.1.2 Features

Simple API. Training and querying agents with Stable-Baselines3 is straightforward, requiring only a few lines of code. This simplicity enables researchers to easily integrate the baseline algorithms and components into their experiments (e. g. [NZSL19, GDW+20, MYY+23, BPB+24]) and apply RL to novel tasks and environments, such as autonomous drone racing [SRM+23] and controlling electric motors [TBKW20]. Fig. 3.1 illustrates the ease of use.

Documentation. SB3 provides extensive documentation of its code API[3]. It includes a user guide with concrete examples, a Colab notebook-based RL tutorial[4], general tips for running RL experiments, and a developer guide. We also pay close attention to questions and uncertainties from SB3 users and update the documentation to address them.

[2] An upgrade guide is available at `https://stable-baselines3.readthedocs.io/en/master/guide/migration.html`

[3] `https://stable-baselines3.readthedocs.io/en/master/`

[4] `https://github.com/araffin/rl-tutorial-jnrr19`, running directly in the browser.

High-Quality Implementations. Algorithms are verified against published results by comparing agent learning curves[5]. All functions are typed, documented, and covered by unit tests (currently 96% code coverage). Continuous integration ensures that modifications pass unit tests, type checking, and code style validation.

Comprehensive. Stable-Baselines3 contains the following state-of-the-art on- and off-policy algorithms, commonly used as experimental baselines: ARS [MGR18], A2C [MBM+16], DDPG [LHP+16], DQN [MKS+15], HER [AWR+17], PPO [SWD+17], SAC [HZAL18], TRPO [SLA+15], TD3 [FvHM18] and QR-DQN [DRBM18]. In addition, SB3 provides several algorithm-independent features. We support logging to CSV files and TensorBoard. Users can log custom metrics and modify training via user-provided callbacks. To speed up training, we support parallel (or "vectorized") environments. To simplify training, we implement common environment wrappers, such as preprocessing Atari observations to match the original DQN experiments [MKS+15].

Stable-Baselines3 Contrib. Experimental features are implemented in a separate contrib repository [RHE+20], allowing Stable-Baselines3 to maintain a stable and compact core while providing the latest features, like Truncated Quantile Critics (TQC) [KSGV20]. Contrib implementations undergo the same rigorous review process to ensure trustworthiness.

3.1.3 Comparison to Related Software

Most RL libraries are designed for experienced researchers, requiring expert knowledge to use [WZD+20, HSA+20, FNKI21, CMG+18, GKR+18, GCL+18, SA19, Kol18]. Few RL libraries offer more than brief API documentation [gc19, LLN+18, KSF17, GKR+18], and some are notoriously difficult to understand[6]. In contrast, Stable-Baselines3 is designed to be easy to use, with extensive documentation and tutorials.

The previous version of Stable-Baselines3, Stable-Baselines2, has its roots in OpenAI Baselines [DHK+17], but the two codebases have diverged significantly (see PR #481). SB3 is a complete rewrite of Stable-Baselines2 in PyTorch, building on the improvements and new algorithms from SB2 while further enhancing code quality (e. g. cleaner codebase, better test coverage, type hints). Compared to Baselines, SB3 is fully documented, commented, tested, and features five additional algorithms (ARS, QR-DQN, SAC, TD3, TQC) and many additional features (e. g. dictionary observation support, callbacks, evaluation with multiple environments, environment checker). The only legacy features of OpenAI Baselines are the code structure (one folder per algorithm), the use of code-level optimizations, and the environment tools, which have been greatly improved[7] (additional features, bug fixes, comments, documentation and more testing).

Many RL libraries have a modular design [CLNE17, KG17b, HSA+20, gc19], allowing for quick combination of advances from different papers. However, this requires new users to understand the full code structure before making changes. On the other hand, educational implementations like Spinning Up [Ach18] or Clean RL [HDY+22] are self-contained but hard to maintain due to code duplication. SB3 strikes a balance, factoring out widely

[5] For example, issue #48 or issue #49.

[6] OpenAI Baselines [DHK+17], see `https://www.reddit.com/r/MachineLearning/comments/95ft1j/`. This was a major starting point for Stable-Baselines2 [HRE+18]

[7] As an example, one can compare "VecNormalize" in OAI Baselines vs SB3.

used components (like replay buffers) while minimizing the amount of code that needs to be understood to modify an algorithm.

Since an exhaustive comparison to all RL libraries is not possible, in Table 3.1 we compare SB3 to a subset of other active or popular libraries, focusing on quality of implementation and openness to new users.

RLlib [LLN+18] scores highly in the table, but targets a different use-case than SB3. While SB3 prioritizes simplicity and reliability, RLlib [LLN+18] focuses on scalability and support for distributed training. RLlib's versatility comes at the cost of a larger and more complex codebase.

Overall, we find that SB3 compares favorably to other libraries in terms of documentation, testing, and activity.

	SB3	OAI Baselines	PFRL	RLlib	Tianshou	Acme	Tensorforce
Backend	PyTorch	TF	PyTorch	PyTorch/TF	PyTorch	Jax/TF	TF
User Guide / Tutorials	✓/ ✓	✗/ –	–/ ✓	✓/ ✓	–/ ✓	–/ ✓	✓/ –
API Documentation	✓	✗	✓	✓	✓	✗	✓
Benchmark	✓	✓	✓	✓	–	–	–
Pretrained models	✓	✗	✓	✗	✗	✗	✗
Test Coverage	96%	49%	?	?	94%	74%	81%
Type Checking	✓	✗	✗	✓	✓	✓	✗
Issue / PR Template	✓	✗	✗	✓	✓	✗	✗
Last Commit (age)	< 1 week	> 4 years	> 1 month	< 1 week	< 1 week	< 1 month	> 1 month
Approved PRs (6 mo.)	75	0	13	222	85	5	7

Table 3.1: Comparison of SB3 to a representative subset of active or popular RL libraries. **Key**: – means that the feature is only partially present; OAI: OpenAI; TF: TensorFlow; PR: Pull Request.

3.2 SBX: A Faster Version of SB3

While Stable-Baselines3 offers extensive features and flexibility using the PyTorch library, SBX is designed to be a faster version with a more limited scope but sharing the same interface. To achieve significant speedup (SBX is up to 20 times faster than SB3), SBX uses the Jax library [BFH+18] and its just-in-time (JIT) compilation capabilities. More specifically, SBX compiles the policy updates that are the bottleneck of SB3.

The faster implementations in SBX have two main advantages. First, experiments can be run much more quickly as shown in Fig. 3.2 (e. g. 10 minutes to run one experiment in simulation instead of two hours). Second, more gradient updates can be performed in the same amount of time, resulting in improved sample efficiency. This is the main idea behind the DroQ [HIH+22] algorithm, which is up to ten times more sample efficient than other model-free algorithms. This improved sample efficiency is particularly relevant when learning directly on real robots.

Importantly, SBX implementations share the same interface and have been benchmarked against SB3 implementations. SBX also reuses significant components from SB3.

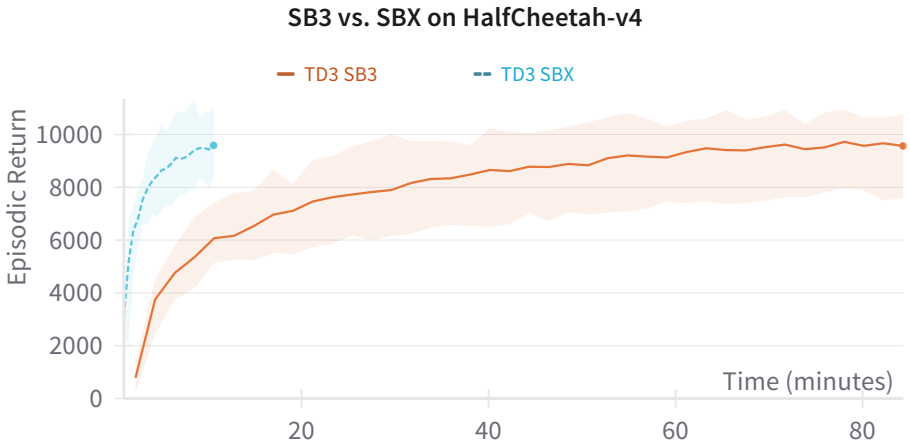


Figure 3.2: Comparison of runtime between SB3 and SBX on the HalfCheetah environment for the TD3 algorithm. Here, SBX is around 10x faster than SB3.

3.3 RL Zoo: A Training Framework for Reproducible RL

RL Baselines Zoo [Raf18, Raf20] is a training framework based on SB3 and SBX. SB3 and SBX provide algorithms and components such as replay buffers and callbacks; RL Zoo complements these libraries with scripts for training, evaluating agents, tuning hyperparameters, visualizing results, and recording videos. Fig. 3.3 shows the main features of the RL Zoo.

In addition, the RL Zoo includes a collection of pre-trained reinforcement learning agents along with tuned hyperparameters for a large variety of tasks, including PyBullet environments [CB21] and Atari games. This allows to quickly assess performance and serve as a starting point for exploring new tasks.

```
# Train an SAC agent on the Pendulum task using tuned hyperparameters,
# evaluate the agent every 1k steps and save a checkpoint every 10k steps
# Pass custom hyperparameters to the algo/env (here gravity=9.8)
python -m rl_zoo3.train --algo sac --env Pendulum-v1 --eval-freq 1000 \
    --save-freq 10000 -params train_freq:2 --env-kwargs g:9.8

# Plot the learning curve
python -m rl_zoo3.cli all_plots -a sac -e Pendulum-v1 -f logs/

# Load and evaluate a trained agent for 1000 steps
# optionally, you can also load a checkpoint using --load-checkpoint
python -m rl_zoo3.enjoy --algo sac --env Pendulum-v1 -n 1000

# Tune the hyperparameters of ppo on BipedalWalker-v3 with a budget of 50 trials
# using 2 parallel jobs, a TPE sampler and median pruner
python -m rl_zoo3.train --algo ppo --env BipedalWalker-v3 -optimize --n-trials 50 \
    --n-jobs 2 --sampler tpe --pruner median
```

Figure 3.3: Using RL Baselines3 Zoo to train, evaluate and save checkpoints; plot results; load, evaluate and render a trained agent; and perform hyperparameter tuning.

The RL Zoo adheres to best practices for RL experimentation [HIB+18b, PNWW23], ensuring reproducible experiments and allowing the user to focus on defining their task.

Each time an experiment is run, the RL Zoo saves all relevant information in a dedicated folder, making it easy to replicate and compare results. This includes the algorithm hyperparameters, the command line arguments, and the arguments passed to the environment (e. g. when trying variations of a task).

Thanks to its integration with platforms like *Weight&Biases* and *HuggingFace*, RL Zoo results can be easily aggregated and compared, the Open RL Benchmark [HGF+24] being an example.

Overall, the RL Baselines3 Zoo is a key component for having reliable and reproducible experiments, which is critical when learning directly in the real world.

3.4 Best Practices for Testing and Implementing RL Algorithms

To conclude this chapter, we will examine two aspects that distinguish SB3 and its associated software suite. First, we will present in more detail the type of tests we use for all tools (SB3, SBX, and RL Zoo), and then dive into the steps required to reliably implement a new RL algorithm. The best practices presented here use RL for illustration, but they can also be applied to other domains.

3.4.1 Automated Tests for RL Software

```python
# Training budget (cap the max number of iterations)
N_STEPS = 1000

def test_performance_ppo() -> None:
    agent = PPO("MlpPolicy", "CartPole-v1").learn(N_STEPS)
    # Evaluate the trained agent
    episodic_return = evaluate_policy(agent, n_eval_episodes=20)
    # check that the performance is above a given threshold
    assert episodic_return > 90
```

Figure 3.4: An example of a performance test that trains a PPO agent with a small budget and checks that it achieves a performance above a given threshold.

SB3 employs several strategies to ensure reliability. One key approach is to adhere to software engineering best practices and use automated checks.

The automated checks include a formatter that unifies the code style, a linter that scans the code for potential errors (e. g. a variable is defined but not used) and a static type checker that helps ensure code correctness (e. g. checking that the type of a variable passed to a function is the expected one). In addition, SB3 has a large collection of tests, which translates into high code coverage (percentage of code covered by at least one test). Three main types of tests are employed: execution tests, unit tests, and performance tests.

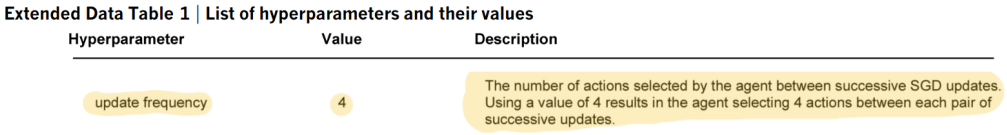
Execution tests (or "run tests") check that the code executes without errors (the different component of the code can work together without crashing), while unit tests validate the behavior of specific features (e. g. for a given input, a function must return an expected output). Performance tests go beyond typical runtime tests by evaluating whether the algorithm demonstrates actual learning progress, rather than random behavior. For

example, as shown in Fig. 3.4, testing a PPO implementation with a small training budget in a simple environment can help detect regressions by setting a performance threshold.

3.4.2 Implementing a New Algorithm

Implementing a new RL algorithm can be a difficult task [HDR+22, HGF+24], but following a systematic approach is the key to success.

Understanding the Algorithm



Extended Data Table 1 | List of hyperparameters and their values

Hyperparameter	Value	Description
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.

Figure 3.5: An excerpt from the appendix of the DQN[MKS+15] paper that mentions the "update frequency" parameter, which has a great impact on the learning process.

The first step in implementing a new algorithm is to thoroughly read the original research paper. It is essential to pay close attention to details that may significantly impact implementation and performance. Examining the appendix and supplementary materials can reveal important hyperparameters or implementation details that may not be explicitly mentioned in the main body of the paper. For example, in the Deep Q-Network (DQN) paper [MKS+13], a detail about network update frequency is only mentioned in the appendix (the excerpt is displayed in Fig. 3.5).

Reviewing Existing Implementations

```
342     class RolloutBuffer(BaseBuffer):
402         def compute_returns_and_advantage(self, last_values: th.Tensor, dones: np.ndarray) -> None:
425             for step in reversed(range(self.buffer_size)):
426                 if step == self.buffer_size - 1:
427                     next_non_terminal = 1.0 - dones
428                     next_values = last_values
429                 else:
430                     next_non_terminal = 1.0 - self.episode_starts[step + 1]
431                     next_values = self.values[step + 1]
432                 delta = self.rewards[step] + self.gamma * next_values * next_non_terminal - self.values[step]
433                 last_gae_lam = delta + self.gamma * self.gae_lambda * next_non_terminal * last_gae_lam
434                 self.advantages[step] = last_gae_lam
435             # TD(lambda) estimator, see Github PR #375 or "Telescoping in TD(lambda)"
436             # in David Silver Lecture 4: https://www.youtube.com/watch?v=PnHCvfgC_ZA
437             self.returns = self.advantages + self.values
```

Figure 3.6: An excerpt from SB3 codebase that shows the computation of the TD target for PPO value function using TD(λ) estimator. If not careful, this type of implementation detail can be easily missed.

Reviewing existing implementations, especially the original implementation by the authors of the algorithm, is crucial. This step helps uncover undocumented tricks and implementation details that can affect performance. For instance, the Proximal Policy

Optimization (PPO) [SWD+17] code reveals the use of a TD(λ) estimator for the value function [SML+15] instead of the Monte Carlo estimator mentioned in the paper (see Fig. 3.6). This subtle but important difference can be easily overlooked without careful examination of the code.

Basic Implementation and Validation

After understanding the details of the algorithm, the next step is to create a basic implementation that exhibits some "sign of life" on a simple toy problem. The goal is to ensure that the algorithm shows some learning progress rather than behaving randomly. Using a toy problem allows for faster iteration and easier debugging. For example, to verify the functionality of a PPO implementation with memory (PPO with LSTM), it can be tested on a modified Pendulum environment [TTK+23], where the velocity observation is removed. This modification makes it impossible for a standard PPO implementation to solve the task, while the PPO with LSTM implementation should successfully learn to control the pendulum, demonstrating that the memory component of the algorithm works as intended.

Step-by-Step Validation

```python
import numpy as np
# Demonstration of automatic broadcasting
batch_size = 64
# shape: (64,)
rewards = np.ones((batch_size,))
# shape: (64, 1)
current_q_values = np.zeros((batch_size, 1))
# auto broadcast, shape: (64, 64)!
td_target = current_q_values + rewards
```

Figure 3.7: An example of unexpected result due to automatic broadcast: when adding two vectors together, the output is a matrix.

Step-by-step validation is another aspect of the implementation. This involves logging relevant values, such as the mean and maximum Q values, to monitor the algorithm's progress and identify potential problems. Using a debugger to step through the code and inspect the variables allows to catch subtle errors such as unintended broadcasting in NumPy [HMvdW+20]. For example, adding a vector to a single-column matrix can cause unexpected broadcasting, resulting in a matrix output instead of a vector (as shown in Fig. 3.7). While the code may run without error, this broadcasting problem can significantly affect the results.

Visualizing the behavior of the trained agent is another validation technique. Observing how the agent interacts with the environment can often provide more insight than simply analyzing metrics.

Thorough Evaluation

Once the implementation seems reasonably functional, the next step is to validate it on known environments with increasing complexity. This allows for a more thorough evaluation of the algorithm's capabilities and helps identify potential issues related to exploration or other aspects of the algorithm's performance [CSO18, FLO+22]. Starting with simpler environments and gradually progressing to more complex ones provides a systematic way of assessing the robustness of the implementation (following Section 2.4).

3.5 Conclusion

In this chapter, we introduced two software tools that facilitate the application of RL to real robots: SB3 and SBX. Stable-Baselines3 provides core features and well-tested implementations, while SBX extends SB3 and significantly improves its performance, allowing for faster iteration and the use of more sample-efficient algorithms. These two software tools offer a reliable foundation for training RL controllers, enabling users to focus on defining the RL task rather than worrying about implementation details. Furthermore, the RL Zoo toolkit ensures reproducibility and adherence to best practices when running the experiments.

This chapter is probably the one with the greatest impact. Since its release, SB3, SBX and the RL Zoo have been used by students, educators, researchers, and engineers who want to apply reinforcement learning. The software has been integrated into Deep RL courses [SS23][8] and is widely used by students [RFK+23][9]. Researchers from various fields have employed it, from video games [Whi23] to large language models [RAB+23], particle accelerators at CERN [VGK+23], climate change problems [LCNB22], biped robots [Car22] or autonomous drone racing [SSKS21, SRM+23]. Finally, it was also used to develop new RL algorithms [BPB+24].

[8] See also `https://edu.epfl.ch/coursebook/en/legged-robots-MICRO-507` and `https://www.kaggle.com/code/alexisbcook/deep-reinforcement-learning`

[9] See also `https://github.com/DLR-RM/stable-baselines3/network/dependents`

CHAPTER 4

Smooth Exploration with generalized State-Dependent Exploration (gSDE)

Most RL practitioners only train agents in simulation [TET12, MBT+18, CB21, MWG+21, TTK+23]. Among the few who apply learned controllers on actual robots, the majority follow the same simulation-to-reality approach [ICT+18, AAC+19, ZQW20, SGFH21, SPB22, KFPM21, MLH+22, RHRH22, ZFW+23]. Training in simulation hides some weaknesses of DRL algorithms that make them unsuitable for real robots. One problem is the default exploration method for continuous control, which relies on adding noise at each step, illustrated to the left in Fig. 4.1. It can be very effective in simulation [DCH+16, ABC+20, FvHM18, PALvdP18, HLD+19] but poses safety risks and is ineffective for real systems.

This chapter contributes a simple alternative to the standard step-based exploration, providing a smoother exploration strategy. Our approach enables the direct training of RL agents in the real-world, tackling challenge i. (Exploration-Induced Wear and Tear), without relying on heuristics like low-pass filters. We integrate the proposed method into Stable-Baselines3 and use the software suite introduced in Chapter 4 to run all the experiments.

4.1 Introduction

When it comes to experiments on real robots, the standard white noise exploration, or unstructured exploration, has many drawbacks. These limitations have been consistently highlighted in the literature [RFS08, KP09, RSS+10, SS13, DNP+13]:

1. Sampling independently at each step can result in shaky behavior [MMMS21], leading to noisy and jittery trajectories.
2. The jerky motion patterns generated by this approach can cause damage to the motors on a real robot, leading to increased wear and tear.
3. A real system acts as a low-pass filter, which means that successive perturbations can cancel each other out, resulting in poor exploration. This is particularly problematic when the control frequency is high [KMVB19].

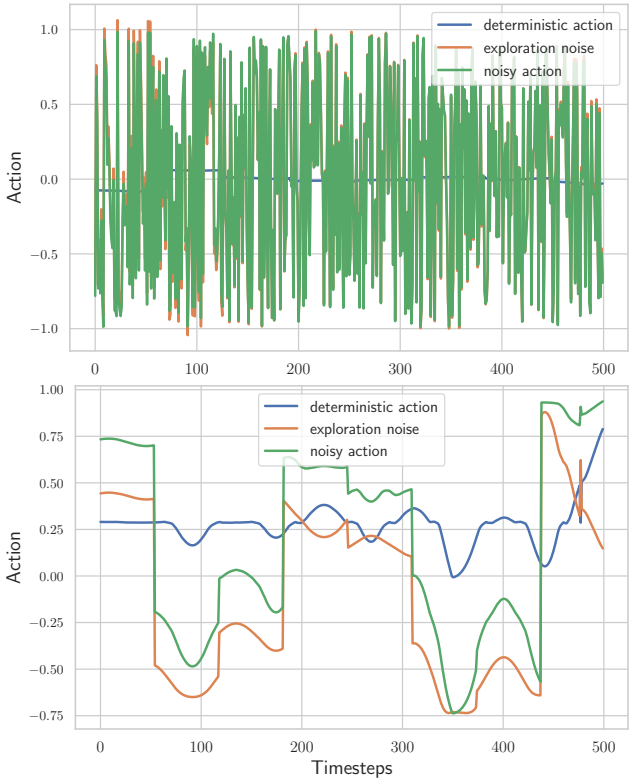


Figure 4.1: Comparison of exploration methods. Top: Unstructured exploration (step-based exploration) producing high-frequency noise, commonly used in simulated DRL. Bottom: Our proposed method, *g*SDE, provides smooth and consistent exploration.

4. Unstructured exploration can lead to a large variance that grows with the number of time-steps [KP09, RSS+10, SS13]

In practice, we have observed all of these drawbacks on three real robots, including the tendon-driven robot *David* Section 2.3.1, which is the main experimental platform used in this chapter.

In robotics, multiple solutions have been proposed to counteract the inefficiency of unstructured noise. These include correlated noise [HZH+18, KMVB19], low-pass filters [HHZ+18, HXT+20], action repeat [NAW+20] or lower level controllers [HHZ+18, KHJ+19]. A more principled solution is to perform exploration in parameter space, rather than in action space [PHD+17, PS18]. This approach usually requires fundamental changes in the algorithm, and is harder to tune when the number of parameters is high.

State-Dependent Exploration (SDE) [RFS08, RSS+10] was proposed as a compromise between exploring in parameter and exploring in action space. SDE replaces the sampled noise with a state-dependent exploration function that returns the same action for a given state during an episode. This results in smoother exploration and less variance per episode.

To the best of our knowledge, no Deep RL algorithm has yet been successfully combined with SDE. We suspect this is because the problem it solves – shaky, jerky motion – is not as noticeable in simulation, which is the current focus of the community.

In this chapter, we aim at reviving interest in SDE as an effective method for addressing exploration issues that arise from using independently sampled Gaussian noise on real

robots. Our concrete contributions, which also determine the structure of the chapter, are:

1. Highlighting the issues with unstructured Gaussian exploration (Section 4.1).
2. Adapting SDE to Deep RL algorithms, and tackling some issues of the original formulation (Sections 4.3.2 and 4.4).
3. Evaluate the different approaches with respect to the compromise between smoothness and performance, and show the impact of the noise sampling interval (Sections 4.5.1 and 4.5.2).
4. Successfully applying DRL directly on three real robots: a tendon-driven robot, a quadruped and an RC car, without need of a simulator or filters (Section 4.5.3).

4.2 Related Work

Exploration is a key topic in reinforcement learning [SB18a]. It has been studied extensively in the discrete case, and most recent work still focuses on discrete actions [OBPVR16, BSO+16, OAC18, FAP+18].

For continuous control, several papers tackle the problem of unstructured exploration by introducing correlated noise. [KMVB19] use an autoregressive process with variables to control exploration smoothness. Similarly, [vHTP17] employ a temporal coherence parameter to interpolate between step- or episode-based exploration, using a Markov chain to correlate the noise. This approach requires a history, altering the problem definition.

Exploring in parameter space [KP09, SOR+10, RSS+10, SS13, SS19] is an orthogonal approach that also solves some issues of unstructured exploration. It has been successfully applied to real robots, but relies on motor primitives [PS08, SS13], which requires initial demonstrations. [PHD+17] adapted parameter exploration to DRL by defining a distance in the action space and applying layer normalization to handle high-dimensional space.

Population-based algorithms like Evolution Strategies (ES) and Genetic Algorithms also explore parameter space. With massive parallelization, they compete with RL in training time but are sample inefficient [SMC+17]. [PS18] combines ES exploration with RL gradient updates to mitigate this inefficiency. While powerful, this approach introduces numerous hyperparameters and significant computational overhead.

Smooth control is essential for real robots but often overlooked in DRL. [MMMS21] incorporated continuity and smoothing loss into RL algorithms, achieving smooth controllers that reduce energy consumption at test time on real robots. However, their method does not address smooth exploration during training, limiting their approach to training in simulation.

4.3 Exploration for Continuous Control

4.3.1 Exploration in Action or Policy Parameter Space

In the case of continuous actions, the exploration is commonly done in the *action space* [SLA+15, LHP+16, MBM+16, SWD+17, HTAL17, FvHM18]. A noise vector ϵ_t is independently sampled from a Gaussian distribution at each time-step, and then added to the policy output as follows:

$$\mathbf{a}_t = \mu(\mathbf{s}_t; \theta_\mu) + \epsilon_t, \qquad \epsilon_t \sim \mathcal{N}(0, \sigma^2). \tag{4.1}$$

Here, $\mu(\mathbf{s}_t, \theta_\mu)$ is the deterministic policy, and $\pi(\mathbf{a}_t|\mathbf{s}_t) \sim \mathcal{N}(\mu(\mathbf{s}_t, \theta_\mu), \sigma^2)$ is the resulting stochastic policy used for exploration. The parameters of the deterministic policy are denoted by θ_μ. For simplicity, we consider only Gaussian distributions with diagonal covariance matrices, so σ is a vector with the same dimension as the action space $\mathcal{A}$.

Alternatively, the exploration can be performed in the policy *parameter space* [RSS+10, PHD+17, PS18]. This involves adding a perturbation ϵ to the policy parameters θ_μ at the beginning of an episode:

$$\mathbf{a}_t = \mu(\mathbf{s}_t; \theta_\mu + \epsilon), \qquad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{4.2}$$

While this approach can lead to more consistent exploration, it becomes increasingly challenging as the number of parameters grows [PHD+17].

4.3.2 State-Dependent Exploration

*State-Dependent Exploration (*SDE*)* [RFS08, RSS+10] is an intermediate approach that adds noise to the deterministic action $\mu(\mathbf{s}_t)$ as a function of the state $\mathbf{s}_t$. At the beginning of an episode, the parameters θ_ϵ of this exploration function are drawn from a Gaussian distribution. The resulting action $\mathbf{a}_t$ is given by:

$$\mathbf{a}_t = \mu(\mathbf{s}_t; \theta_\mu) + \epsilon(\mathbf{s}_t; \theta_\epsilon), \qquad \theta_\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{4.3}$$

This episode-based exploration is smoother and more consistent than unstructured step-based exploration. As a result, during an episode, the action $\mathbf{a}$ for a given state $\mathbf{s}$ will be the same, rather than oscillating around a mean value.

In the rest of this chapter, we drop the time subscript t to avoid overloading the notation, i. e. we write $\mathbf{s}$ instead of $\mathbf{s}_t$.

In the case of a linear exploration function $\epsilon(\mathbf{s}; \theta_\epsilon) = \theta_\epsilon \mathbf{s}$, it can be shown that the action element $\mathbf{a}_j$ is normally distributed [RFS08]:

$$\pi_j(\mathbf{a}_j|\mathbf{s}) \sim \mathcal{N}(\mu_j(\mathbf{s}), {\hat{\sigma}_j}^2) \tag{4.4}$$

where $\hat{\sigma}$ is a diagonal matrix with elements $\hat{\sigma}_j = \sqrt{\sum_i (\sigma_{ij}\mathbf{s}_i)^2}$.

Since we know the policy distribution, we can obtain the derivative of the log-likelihood $\log \pi(\mathbf{a}|\mathbf{s})$ with respect to the variance σ:

$$\frac{\partial \log \pi(\mathbf{a}|\mathbf{s})}{\partial \sigma_{ij}} = \sum_k \frac{\partial \log \pi_k(\mathbf{a}_k|\mathbf{s})}{\partial \hat{\sigma}_j} \frac{\partial \hat{\sigma}_j}{\partial \sigma_{ij}} \tag{4.5}$$

$$= \frac{\partial \log \pi_j(\mathbf{a}_j|\mathbf{s})}{\partial \hat{\sigma}_j} \frac{\partial \hat{\sigma}_j}{\partial \sigma_{ij}} \tag{4.6}$$

$$= \frac{(\mathbf{a}_j - \mu_j)^2 - {\hat{\sigma}_j}^2}{{\hat{\sigma}_j}^3} \frac{\mathbf{s}_i^2 \sigma_{ij}}{\hat{\sigma}_j} \tag{4.7}$$

This can be easily plugged into the likelihood ratio gradient estimator [Wil92], allowing σ to be adjusted during training. SDE is thus compatible with standard policy gradient methods, while addressing most of the shortcomings of unstructured exploration.

4.4 Generalized State-Dependent Exploration

The original formulation of SDE has several limitations that follow from Eqs. (4.4) and (4.5). In particular:

i The noise remains constant throughout an episode, which can limit exploration in long episodes [vHTP17].

ii The policy variance $\hat{\sigma_j} = \sqrt{\sum_i (\sigma_{ij}\mathbf{s}_i)^2}$ depends on the state space dimension, requiring problem-specific adjustments to the initial σ.

iii The exploration noise is linearly dependent on the state, limiting its expressiveness.

iv The state must be normalized, as the gradient and noise magnitude depend on the state magnitude.

To address these limitations and adapt SDE to Deep RL algorithms, we propose two improvements:

1. We sample the exploration function parameters θ_ϵ every n steps, rather than every episode.

2. Instead of using the state $\mathbf{s}$, we use policy features $\mathbf{z}_\mu(\mathbf{s}; \theta_{\mathbf{z}_\mu})$ (the last layer before the deterministic output $\mu(\mathbf{s}) = \theta_\mu \mathbf{z}_\mu(\mathbf{s}; \theta_{\mathbf{z}_\mu})$) as input to the noise function $\epsilon(\mathbf{s}; \theta_\epsilon) = \theta_\epsilon \mathbf{z}_\mu(\mathbf{s})$.

Sampling θ_ϵ every n steps overcomes limitation i and provides a unifying framework [vHTP17] that encompasses both unstructured exploration ($n = 1$) and original SDE ($n =$ episode_length). Although this formulation follows the description of DRL algorithms that update their parameters every m steps, the influence of this parameter on smoothness and performance has been overlooked until now.

Using *policy features* addresses issues ii, iii, and iv. The relationship between state $\mathbf{s}$ and noise ϵ becomes non-linear, and the variance of the policy depends only on the network architecture. This makes gSDE more task-independent, since the network architecture typically remains constant. It also reduces the number of parameters and computations required for large state spaces, such as images. The number of parameters and operations is determined only by the size of the last layer and the action dimension, no longer by the size of the state space. Therefore, this formulation is more general and includes the original SDE description when the state is used as an input to the noise function or when the policy is linear.

We refer to the resulting approach as *generalized* State-Dependent Exploration (gSDE).

4.5 Experiments

In this section, we study gSDE to answer the following questions:

- How does gSDE compares to the original SDE? What is the impact of each proposed change?
- How does gSDE compares to other types of exploration noise in terms of the tradeoff between smoothness and performance?
- How does gSDE performs on a real robot?

4.5.1 Compromise Between Smoothness and Performance

Experimental Setup To compare the performance and smoothness of gSDE with other exploration methods, we use four locomotion tasks from the PyBullet package [CB21]: HalfCheetah, Ant, Hopper, and Walker2D. We focus on the SAC algorithm, as it will be used on a real robot.

To evaluate smoothness, we define a continuity cost $\mathcal{C} = 100 \times \mathbb{E}_t \left[(\frac{\mathbf{a}_{t+1} - \mathbf{a}_t}{\Delta \mathbf{a}_{\max}})^2 \right]$, ranging from 0 (constant output) to 100 (action jumps from one boundary to another at every step). The training continuity cost $\mathcal{C}_{\text{train}}$ serves as a proxy for the robot's wear-and-tear.

We compare the performance of the following configurations:

a) No exploration noise

b) Unstructured Gaussian noise (original SAC implementation)

c) Correlated noise (Ornstein–Uhlenbeck process [UO30], $\sigma = 0.2$, labeled OU noise)

d) Adaptive parameter noise [PHD+17] ($\sigma = 0.2$)

e) gSDE

To isolate exploration noise from parameter update noise and to better simulate a real robot scenario, gradient updates are applied only at the end of each trial.

We fix the budget to one million steps and report the average score over 10 runs, along with the average continuity cost during training and its standard error. For each run, we test the learned policy on 20 evaluation episodes every 10000 steps, using the deterministic controller $\mu(\mathbf{s}_t)$. Regarding the implementation, we use the software tools presented in Chapter 3: Stable-Baselines3 implementations [RHG+21] together with the RL Zoo training framework [Raf20].

Algorithm	HalfCheetah		Ant		Hopper		Walker2D	
SAC	Return ↑	$\mathcal{C}_{\text{train}}$ ↓	Return ↑	$\mathcal{C}_{\text{train}}$ ↓	Return ↑	$\mathcal{C}_{\text{train}}$ ↓	Return ↑	$\mathcal{C}_{\text{train}}$ ↓
w/o noise	2562 +/- 102	**2.6** +/- 0.1	2600 +/- 364	**2.0** +/- 0.2	1661 +/- 270	**1.8** +/- 0.1	2216 +/- 40	**1.8** +/- 0.1
w/ unstructured	**2994** +/- 89	4.8 +/- 0.2	**3394** +/- 64	5.1 +/- 0.1	**2434** +/- 190	3.6 +/- 0.1	2225 +/- 35	3.6 +/- 0.1
w/ OU noise	2692 +/- 68	2.9 +/- 0.1	2849 +/- 267	2.3 +/- 0.0	2200 +/- 53	2.1 +/- 0.1	2089 +/- 25	2.0 +/- 0.0
w/ param noise	2834 +/- 54	2.9 +/- 0.1	3294 +/- 55	**2.1** +/- 0.1	1685 +/- 279	2.2 +/- 0.1	**2294** +/- 40	**1.8** +/- 0.1
w/ gSDE-2	**2987** +/- 85	4.1 +/- 0.2	**3366** +/- 50	4.7 +/- 0.1	**2532** +/- 70	2.8 +/- 0.1	2237 +/- 55	2.8 +/- 0.1
w/ gSDE-4	2798 +/- 41	4.1 +/- 0.2	**3227** +/- 182	3.8 +/- 0.2	2541 +/- 49	2.6 +/- 0.1	**2322** +/- 69	2.6 +/- 0.1
w/ gSDE-8	**2850** +/- 73	4.1 +/- 0.2	**3459** +/- 52	3.9 +/- 0.2	**2646** +/- 45	2.4 +/- 0.1	**2341** +/- 45	2.5 +/- 0.1
w/ gSDE-64	**2970** +/- 132	3.5 +/- 0.1	3160 +/- 184	3.5 +/- 0.1	2476 +/- 99	**2.0** +/- 0.1	**2324** +/- 39	2.3 +/- 0.1
w/ gSDE-Episodic	2741 +/- 115	3.1 +/- 0.2	3044 +/- 106	2.6 +/- 0.1	2503 +/- 80	**1.8** +/- 0.1	**2267** +/- 34	2.2 +/- 0.1

Table 4.1: Detailed results for SAC with various exploration methods on PyBullet environments. We present the mean and standard error of returns and continuity costs over ten runs of one million steps. For each benchmark, we highlight the results of the method(s) with the highest mean return if the difference is statistically significant.

Results Table 4.1 and Fig. 4.2 presents the results on the PyBullet tasks and the trade-off between continuity and performance. Without any noise ("No Noise" in the figure), SAC can partially solve these tasks due to a shaped reward, but it exhibits the highest result variance. Although the correlated and parameter noise achieve lower continuity costs during training, it comes at the expense of performance. gSDE strikes a balance

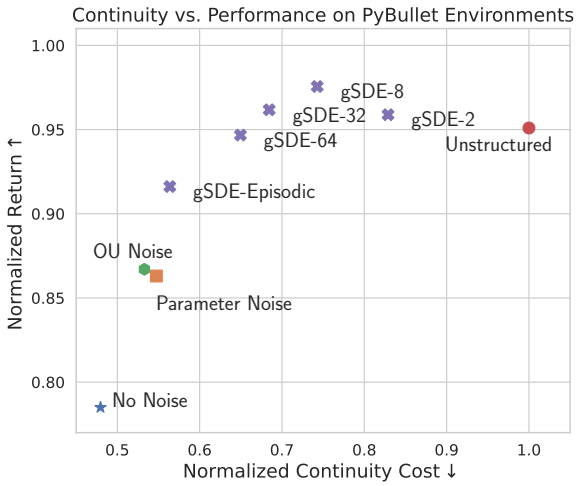


Figure 4.2: Trade-off between normalized return and continuity cost of SAC on four locomotion tasks with various exploration methods. *g*SDE strikes a balance between smoothness and performance, offering a compromise between these two objectives.

between unstructured exploration and correlated noise using the noise repetition parameter n. Specifically, *g*SDE-8 (sampling the noise every $n = 8$ steps) achieves even better performance with a lower continuity cost during training. This balance is ideal for training on a real robot, as it minimizes wear-and-tear during training while maintaining good performance at test time.

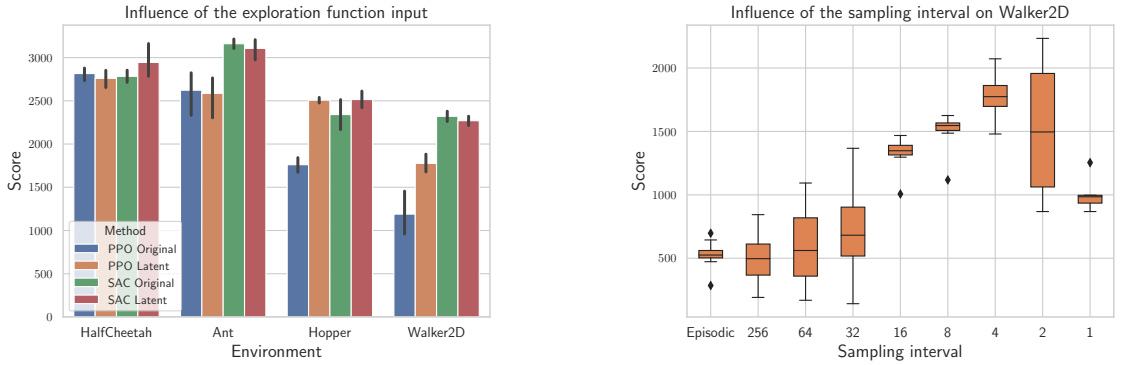
4.5.2 Comparison to the Original SDE

In this section, we examine the contribution of the proposed modifications to the original SDE: sampling the parameters of the exploration function every n steps and using policy features as input to the noise function.

Sampling Interval *g*SDE is an n-step extension of SDE, where n allows interpolation between unstructured exploration ($n = 1$) and the original SDE per-episode formulation. This flexibility provides a balance between smoothness and performance during training (cf.Table 4.1 and Fig. 4.2). Fig. 4.3b demonstrates the importance of this parameter for PPO on the Walker2D task. A large sampling interval results in insufficient exploration during long episodes, while a high sampling frequency ($n \approx 1$) leads to issues discussed in Section 4.1.

Policy features as input Fig. 4.3a presents the impact of changing the input to the exploration function for SAC and PPO. Using policy features ('latent' in the figure) is generally advantageous, particularly for PPO. This approach also requires less tuning and no normalization, as it relies solely on the policy network architecture. In this case, the PyBullet tasks have low-dimensional state spaces, so no environment-specific tuning is necessary. Using features also enables learning directly from pixels, which is not possible in the original formulation.

Compared to the original SDE, the two proposed modifications enhance performance, with the noise sampling interval n having the most significant impact. Fortunately, as



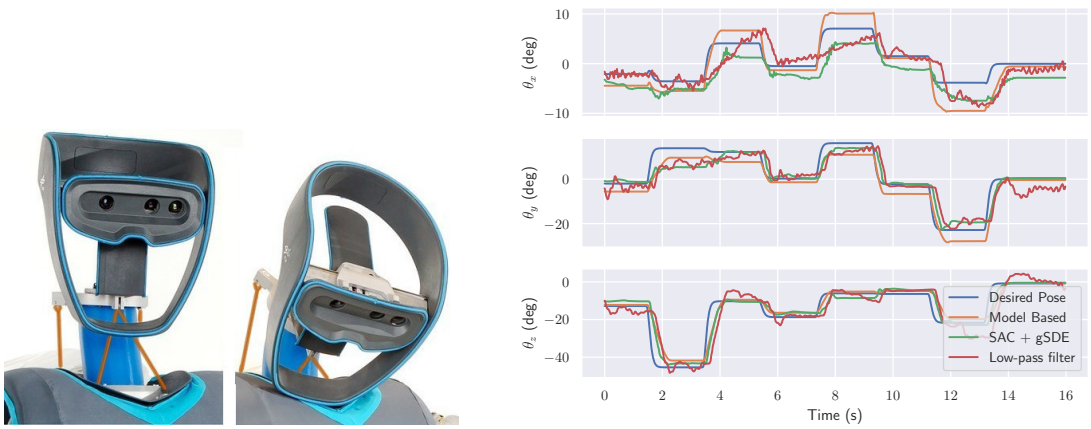
(a) Exploration function input

(b) Sampling interval (PPO on WALKER2D)

Figure 4.3: Impact of *g*SDE modifications on PyBullet tasks. (a) Influence of the input to the exploration function $\epsilon(\mathbf{s}; \theta_\epsilon)$ for SAC and PPO: using latent features from the policy $\mathbf{z}_\mu$ (Latent) is generally better than using the state $\mathbf{s}$ (Original). (b) The sampling interval of the noise function parameters is important for PPO with *g*SDE.

shown in Table 4.1 and Fig. 4.2, this parameter can be chosen relatively freely for SAC.

4.5.3 Learning to Control a Tendon-Driven Elastic Robot



(a) Tendon-driven elastic continuum neck in a humanoid robot

(b) Model-Based controller vs RL controller on the real robot

Figure 4.4: (a) The tendon-driven *David* neck robot [RDF16] used in the experiment, with tendons highlighted in orange. (b) Comparison of tracking performance on an evaluation trajectory: the model-based controller and the RL agent achieve similar results.

Experimental Setup To evaluate the effectiveness of *g*SDE, we apply it to a real-world system: controlling a tendon-driven elastic continuum neck (introduced in Section 2.3.1 and shown in Fig. 4.4a) to achieve a target pose. The control task is challenging due to the nonlinear tendon coupling and deformation of the structure, which requires accurate

modeling. However, this modeling is computationally expensive [DDO17, DCR+19] and relies on assumptions that may not hold in the real system.

The system is underactuated, with only four tendons, and the desired pose is defined by a 4D vector: three angles for rotation (θ_x, θ_y, θ_z) and one for position (x). The input is a 16D vector consisting of measured tendon lengths (4D), current tendon forces (4D), current pose (4D), and target pose (4D). The reward is a weighted sum of two components: negative geodesic distance to the desired orientation and negative Euclidean distance to the target position. The weights are chosen to balance the magnitudes of both components. A small continuity cost is added to reduce oscillations in the trained policy.

The action space is the desired change in tendon forces, limited to 5 N. For safety, tendon forces are clipped between 10 N and 40 N. A trial ends when the agent reaches the desired pose or after 5 s. Success is defined as reaching the desired pose within a 10 mm position threshold and a 5° orientation threshold. The agent sets desired tendon forces at 30 Hz, while a PD controller regulates motor current at 3 kHz.

Results We initially tested unstructured exploration on the robot, but had to halt the experiment early: the high-frequency noise in the commands damaged the tendons and risked breaking them because of friction on the bearings. As a baseline, we trained a controller using SAC with a hand-tuned action smoothing (a 2 Hz cutoff Butterworth low-pass filter) for two hours. We then trained a policy using SAC with gSDE for the same duration.

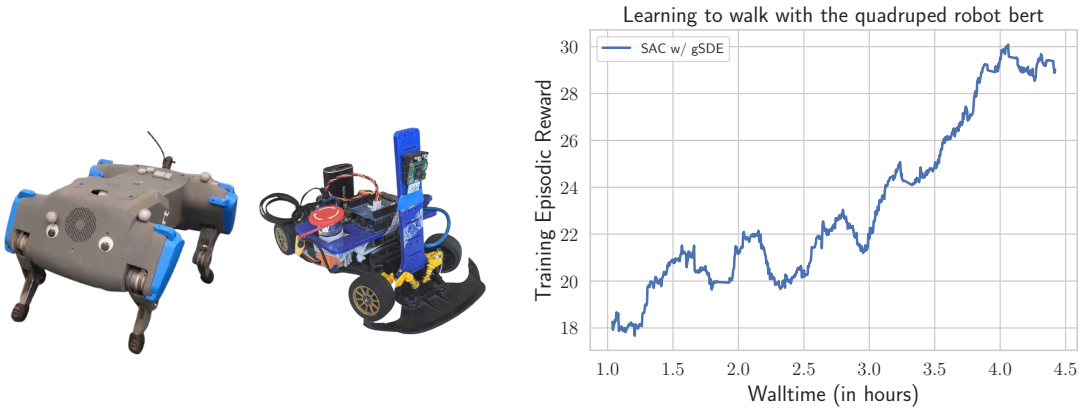
To evaluate the performance of both learned controllers, we compared them to an existing model-based controller (passivity-based approach) from [DDO17, DCR+19] using a predefined trajectory (cf. Fig. 4.4b). The results show that all controllers achieved similar accuracy on the evaluation trajectory (cf. Table 4.2), with mean orientation errors below 3° and position errors below 3 mm. However, the policy trained with the low-pass filter exhibited significantly more jitter than the others. This jitter is quantified as the mean absolute difference in action between two timesteps, referred to as the *continuity cost* in Table 4.2.

	Unstructured noise	gSDE	Low-pass filter	Model-Based
Position error (mm)	N/A	2.65 +/- 1.6	1.98 +/- 1.7	1.32 +/- 1.2
Orientation error (deg)	N/A	2.85 +/- 2.9	3.53 +/- 4.0	2.90 +/- 2.8
Continuity cost (deg)	N/A	**0.20 +/- 0.04**	0.38 +/- 0.07	**0.16 +/- 0.04**

Table 4.2: Comparison of the mean error in position, orientation, and mean continuity cost on the evaluation trajectory. The results show that the model-based and learned controllers achieve comparable performance, while the policy trained with the low-pass filter has a significantly higher continuity cost, indicating increased jitter.

4.5.4 Additional Real Robot Experiments

To showcase the versatility of gSDE, we successfully applied SAC with gSDE to two additional real-world robotics tasks (illustrated in Fig. 4.5a): (1) training an elastic quadruped robot to walk, with the learning curve shown in Fig. 4.5b, (2) learning to drive around



(a) Quadruped and RC car robots

(b) Learning curve on the walking task for the quadruped robot

Figure 4.5: Additional robots that were successfully trained with SAC using *g*SDE in the real world, demonstrating the applicability of the approach to various robotic systems.

a track with an RC car. Notably, both experiments were conducted entirely on the real robots, without relying on simulation or filters, and using the same hyperparameters as for Section 4.5.3.

Learning to Drive a Racing Car in Minutes In this experiment, a small racing car is trained to drive autonomously. The car is equipped with minimal sensors: a camera to capture images of the track and sensor to measure motor speed. This setup presented several challenges, such as variability in lighting, shadows, and the presence of other cars, in addition to constraints on computational power and communication delays.

To address these challenges, we first trained an autoencoder to learn a compact representation of the visual scene. This allows the RL agent to learn from a lower-dimensional input, significantly speeding up the learning process [LDRGF18, KHJ+19, RHT+19]. The action space is also reduced by limiting the steering angle, which helps to achieve smoother driving behavior. The reward function was designed to encourage the car to maximize speed while maintaining smoothness. A safety driver is in charge of terminating the episode before the car crashes. The agent sets steering and throttle commands at 10 Hz, while a low-level controller regulates motor currents at a higher frequency.

The training takes only 5-10 minutes for the car to learn basic driving. The car could successfully navigate the track, even with challenging lighting conditions and obstacles.

Learning to Walk with an Elastic Quadruped Robot In this experiment, the *bert* elastic quadruped was trained to walk. This experiment came with multiple challenges. Ensuring the robot's safety was crucial, as the fast motors and soft springs could cause it to tip over, potentially damaging the robot's case and electronics. Manual resets were required after falls, and communication delays over Wi-Fi further complicated the training process. To overcome these challenges, safety bars were installed to prevent the robot from tipping over, and a basic recovery strategy was implemented to semi-automate resets. A treadmill was also used to help reposition the robot between episodes.

The RL agent received information about its joint positions, torques, acceleration, and

orientation via an inertial measurement unit (IMU) and a gyro sensor. To account for communication delays and promote smooth movements, historical data, such as previous actions and states, were included in the observation space. The action space consisted of six dimensions, representing the desired motor positions for each joint. The reward function was designed to encourage forward movement while maintaining stability. It included a primary reward for forward distance and secondary rewards for walking straight and minimizing jerk.

The training process took several hours (see learning curve in Fig. 4.5b), with the robot starting to move forward after three hours, achieving a basic walking gait after four hours, and refining its movements to become more stable after five to six hours.

4.6 Conclusion

This chapter highlights several problems that arise when using unstructured exploration in Deep RL algorithms for continuous control. These issues prevent DRL algorithms from learning directly on real robots.

To address these issues, we adapt State-Dependent Exploration to Deep RL algorithms (gSDE) by extending the original formulation: we sample the noise every n steps and use learned features as input to the exploration function. gSDE achieves competitive results on several continuous control benchmarks while reducing wear-and-tear during training. An ablation study shows that the noise sampling interval has the most significant impact, balancing performance and smoothness.

In addition to its use on other robots [SHI+23], gSDE has inspired several follow-up works addressing exploration for continuous control [EHPM22, CMVHM23]. It has been extended to high-dimensional action space [CMVHM23], which was part of the winning object manipulation solution of the MyoChallenge 2023[1]. With gSDE, RL algorithms can be applied directly on real robots without additional heuristics (such as low-pass filters) and without damaging the motors during exploration.

In this chapter, we successfully trained policies from scratch on various hardware platforms. In subsequent chapters (Chapter 5 and Chapter 6), we will attempt to build upon these initial efforts by incorporating expert knowledge, for the *David* elastic neck and the *bert* quadruped tasks (see Section 2.3), with the aim of improving policy performance.

[1] See `https://github.com/amathislab/myochallenge-lattice`

CHAPTER 5

Integrating Fault-Tolerant Pose Estimation and RL

In Chapter 4, we discussed how to apply RL in real-world settings, specifically controlling the *David* elastic neck [RDF16] (shown in Fig. 2.9). However, learning from scratch is usually not the best option when training directly on the real robot; it still requires two hours of interactions. In addition, the controller relies on a neck position estimator to achieve the desired pose. Due to the nature of the soft neck, obtaining such estimator is not trivial.

In this chapter, we first contribute a data-driven approach for learning a fault-tolerant neck pose estimator. We then explore different levels of expert knowledge to guide the RL agent, with the goal of reducing training time and improving performance, addressing challenges i. (Exploration-Induced Wear and Tear) ii. (Sample Efficiency) and iv. (Computational Resource Constraints). One option is to reframe the problem as a goal-conditioned task and use hindsight experience replay to relabel experiences, thereby improving sampling efficiency. Finally, we design a feedforward controller by inverting the pose estimator and integrate it with reinforcement learning to close the control loop.

5.1 Fault-Tolerant Pose Estimation

To enable accurate control of *David*'s neck pose (presented in Section 2.3.1) in downstream applications, a reliable pose estimation method is required. However, rigorous models that convert actuator encoder values to end-effector poses are computationally expensive and prone to model parameter uncertainties. Furthermore, for mobile systems, such a pose estimation method should only require an external tracking system for calibration and evaluation.

To address these challenges, we propose a data-driven approach to pose estimation for elastic robots that incorporates uncertainty estimation and error handling. Our main contributions are:

- A fast, six-DoF pose estimation method for tendon-driven continuum mechanisms using a small number of data points (Section 5.1.2).
- An uncertainty estimation technique to detect sensor failure (e. g. slack tendons). This is achieved by building ensembles of estimators and observing their uncertainty,

each ensemble using a subset of the sensor information as input (Section 5.1.2).

- A strategy for dealing with sensor failure by adjusting the pose estimation. When an anomaly is detected, we select the estimators that do not rely on the faulty sensors and continue to accurately predict the pose (Section 5.1.2).
- Experimental validation of these methods on the *David* elastic neck. We demonstrate a significant improvement in pose estimation accuracy and show that reliable predictions can still be made with only three out of eight sensors (Section 5.1.3).

5.1.1 Related Work

Pose estimation for elastic structures. To estimate the pose of elastic structures, there are two primary methods. When geometric and material parameters are accurately known, [JW06] and [CCS09, RW11] have employed geometric or static deformation models, which rely on actuator positions or forces. [DCR+19] proposed a pose estimation technique based on a model for pose-dependent length measurements, leveraging tendon actuation encoder values and deformation-based length sensors. Model-based approaches are sensitive to parameter uncertainty and rely on the assumption of taut tendons, assumptions that can be violated by rapid motion or external contact, leading to performance degradation.

In contrast, data-driven approaches learn direct input-output mappings from measured data, avoiding the need for deformation models. These methods typically use actuation torques or actuator positions as inputs and end-effector pose as the output. In particular, [BDWN07] and [GRC+15] have used neural networks, while [MQS16] used a Gaussian mixture model to map actuator lengths to end-effector poses.

Fault detection in robotic systems. Fault detection in robotic systems [KK18] is a specific instance of the broader field of anomaly detection [CBK09] and can be divided into two main categories: knowledge-based and data-driven approaches.

Knowledge-based approaches rely on the assumption that all faults and their corresponding symptoms are known, enabling causal analysis to identify the fault [HLTB01]. Data-driven approaches, on the other hand, use machine learning techniques to classify normal and abnormal behavior, either in a supervised [HUK+14] or unsupervised manner [HW07, YLS10]. Other data-driven methods include generating nominal behavior data using physical simulators [HKB15] and employing statistical filters, such as Kalman filters [STA16, SRSB12, GDRS00].

Our proposed method is an unsupervised machine learning approach, eliminating the need for labels. Unlike previous approaches that only detect potential failures, we leverage minimal task knowledge to detect and handle faults.

5.1.2 Method

Pose Estimation as a Regression Problem

We formulate the task of predicting the six-DoF pose $\boldsymbol{q}$ given measurements $\tilde{\boldsymbol{l}}$ as a supervised learning problem. For a given estimator f_Θ, with parameters $\Theta \in \mathbb{R}^p$, the goal is to find the parameters Θ that minimize the error between the true pose $\boldsymbol{q}$ and its prediction $\hat{\boldsymbol{q}} = f_\Theta(\tilde{\boldsymbol{l}}) \in \mathbb{R}^6$. This is achieved by minimizing the loss function $\mathcal{L}$:

$$\mathcal{L}(\boldsymbol{q}, \hat{\boldsymbol{q}}) = \|\boldsymbol{q} - \hat{\boldsymbol{q}}\|_2^2 . \tag{5.1}$$

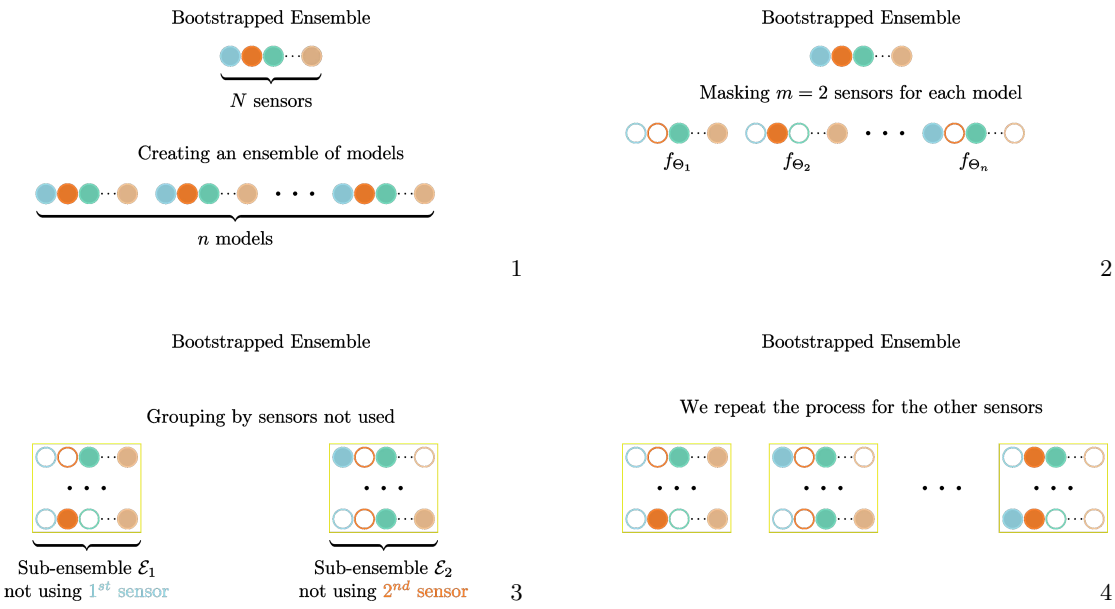


Figure 5.1: Illustration of the ensemble $\mathcal{E}$ and sub-ensembles creation for $m = 2$ masked sensors per estimator. 1-2. We create an ensemble of models, denoted as $\mathcal{E}$, masking $m = 2$ sensors per estimator. 3-4. The estimators are grouped based on the sensor that was not masked, creating sub-ensembles which are used for uncertainty estimation and failure detection.

This is a standard regression problem that can be solved using a variety of techniques [SS15]. In this chapter, we employ linear models $f_\Theta(\tilde{\boldsymbol{l}}) = \Theta^T \tilde{\boldsymbol{l}}$ and second-order polynomial models $f_\Theta(\tilde{\boldsymbol{l}}) = \Theta^T \phi(\tilde{\boldsymbol{l}})$, where ϕ extracts polynomial features, and use least squares to estimate the parameters. We choose these models and methods because they are computationally efficient and do not require hyperparameter tuning or large amounts of data. Furthermore, we found that adding more complexity (e. g. using a neural network or higher-order polynomial) does not improve accuracy.

Uncertainty Estimation using Bootstrapped Ensemble

While querying a single predictor f_Θ is the fastest approach, it does not provide uncertainty estimation, which is essential for detecting and handling failures. We distinguish between two types of uncertainty: aleatoric and epistemic uncertainty [Gal16]. Aleatoric uncertainty arises from sensor noise and is irreducible (unless sensor precision is improved). Epistemic uncertainty, on the other hand, corresponds to the uncertainty in the model, which can be reduced by providing more training data. This type of uncertainty increases with samples that are out of the training distribution [KG17a]. It is this uncertainty that enables failure detection.

We can estimate uncertainty by training an ensemble of n models, denoted as $\mathcal{E} = \{f_{\Theta_1}, f_{\Theta_2}, \dots f_{\Theta_n}\}$, where each model has different parameters Θ_i. For example, this can be achieved by using different parameter initialization for each model.

The uncertainty of the ensemble can be measured by calculating the variance of the predictions. Specifically, the uncertainty for dimension k of the pose $\boldsymbol{q} \in \mathbb{R}^6$ can be estimated using:

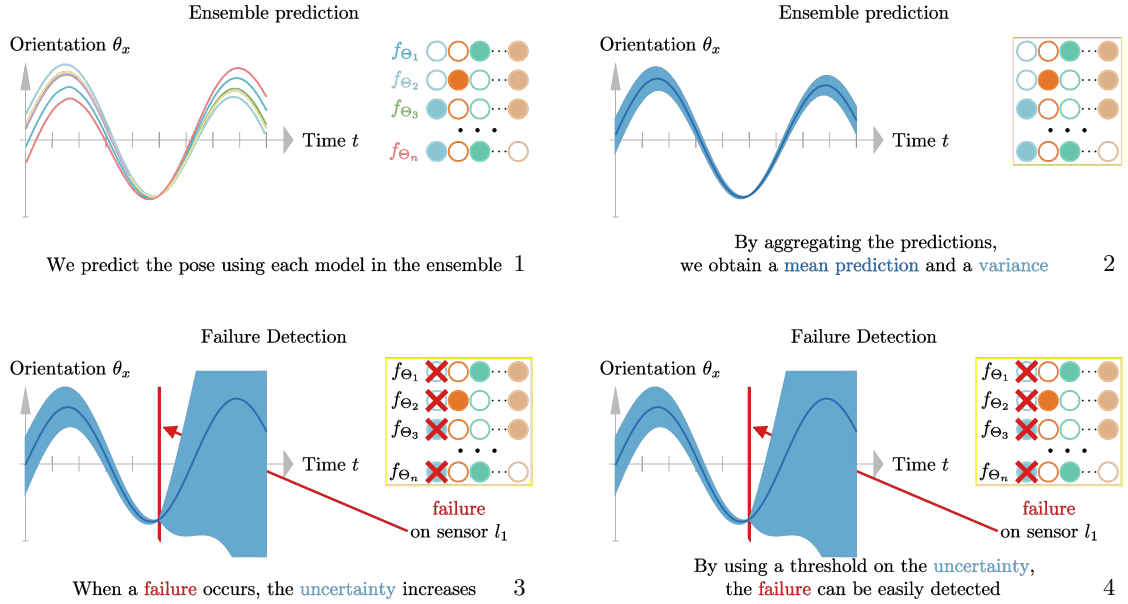


Figure 5.2: Illustration of the failure detection using an ensemble of estimators. 1. Each model in the ensemble $\mathcal{E}$ predicts the pose of the system. 2. The predictions from all models are aggregated to obtain a mean and a variance σ^2. 3-4. A threshold is applied to the variance to detect potential failures. If the variance exceeds the threshold, a failure is detected.

$$\sigma_k^2 = \frac{1}{n}\sum_{i=1}^{n}(f_{\Theta_i}(\tilde{\boldsymbol{l}})_k - \overline{f_\Theta(\tilde{\boldsymbol{l}})}_k)^2 \tag{5.2}$$

where $\overline{f_\Theta(\tilde{\boldsymbol{l}})}$ is the mean of the predictions and the subscript k denotes the k-ith element of the vector.

However, such an ensemble underestimates the epistemic uncertainty, as all the models are trained on the same dataset $\mathcal{D}$. To address this, each model f_{Θ_i} can be trained only on a subset of the training set $\mathcal{D}_i \subset \mathcal{D}$. This way, each model makes different errors, reducing the overconfidence of the ensemble. This technique is known as bootstrapping [Efr92, Bre96], and the resulting ensemble is called a bootstrapped ensemble. The advantage of a bootstrapped ensemble is that any predictor f_{Θ_i} can be used. Fig. 5.1 illustrates the creation of ensembles and sub-ensembles of models.

Failure Detection and Handling

The variance σ_k^2 of the ensemble $\mathcal{E}$ of predictors provides an uncertainty measure. This uncertainty measure can be used to detect failures by applying a threshold ϵ. Specifically, a failure is detected if $\sigma_k^2 > \epsilon$. Fig. 5.2 illustrates how the ensemble can be used to detect a failed sensor.

In our six-DoF pose estimation problem, we have redundant sensor information: there are more sensors than needed, and the system exhibits coupling. Due to the extra sensors, we can detect which sensor(s) failed by grouping the predictions accordingly. This allows accurate pose estimation $\boldsymbol{q}$ even with multiple sensor failures. The key is in how we define

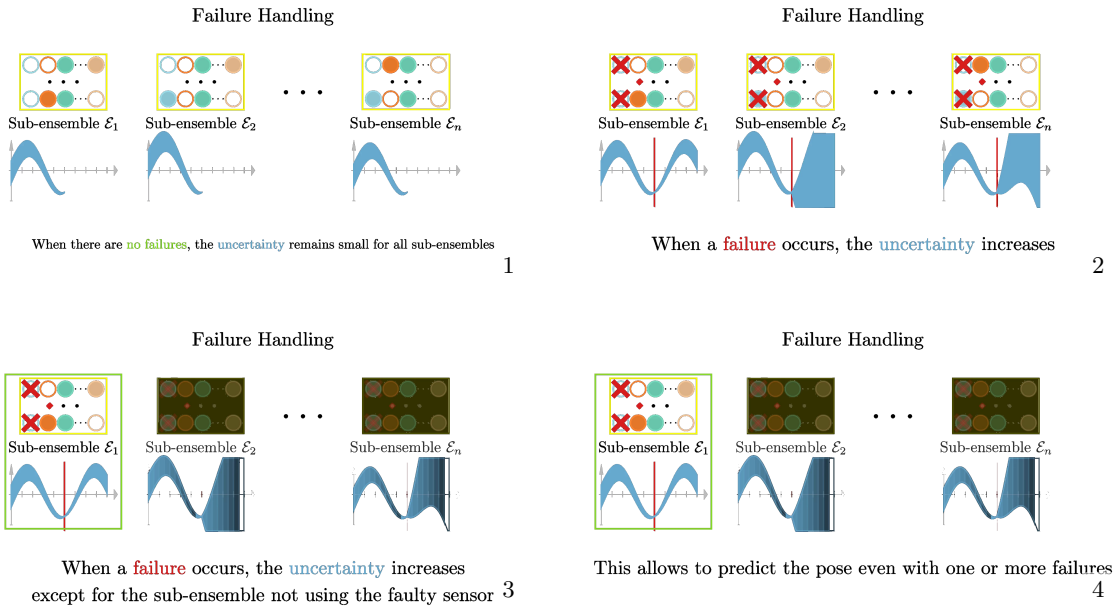


Figure 5.3: Illustration of the failure handling using an ensemble of estimators. 1. Each sub-ensemble predicts the pose, and the uncertainty remains small, indicating that all sensors are functioning correctly. 2-3. A failure occurs, causing the uncertainty to increase for most sub-ensembles. However, one sub-ensemble remains unaffected by the failure, as it does not rely on the faulty sensor. 4. This sub-ensemble continues to predict the pose accurately and with low uncertainty.

the subsets $\mathcal{D}_i \subset \mathcal{D}$ to build the ensemble (cf.Fig. 5.1): we use only part of the sensor measurements $\tilde{\boldsymbol{l}}$ to construct the subsets and then group the estimators into sub-ensembles $\mathcal{E}_l \subset \mathcal{E}$ based on the sensors not used for prediction. For example, masking the first sensor results in the input vector $\tilde{\boldsymbol{l}} = \begin{bmatrix} 0 & \tilde{l}_2 & \tilde{l}_3 & \cdots & \tilde{l}_8 \end{bmatrix}$.

To illustrate this, consider the case where we mask $m = 2$ sensors (out of $N = 8$) as shown in Figs. 5.1 and 5.3. Among the ensemble of trained models $\mathcal{E} = \{f_{\Theta_i}, i = 1 \dots n\}$, one model does not use sensors $\tilde{l}_1$ and $\tilde{l}_2$ (masked sensors are filled with black in Fig. 5.1), another does not use $\tilde{l}_1$ and $\tilde{l}_3$, and so on. Thus, we have a sub-ensemble $\mathcal{E}_{\tilde{l}_1} \subset \mathcal{E}$ composed of estimators that do not use the first sensor $\tilde{l}_1$ for predicting the pose (first sub-ensemble in Fig. 5.1), a second sub-ensemble $\mathcal{E}_{\tilde{l}_2}$ with estimators that do not use the second sensor $\tilde{l}_2$, and so forth. As a result, if one sensor fails as shown in Fig. 5.3, we can still use the predictions from the sub-ensemble that does not use the failing sensor.

To determine if a sub-ensemble $\mathcal{E}_l$ is usable, we compute its variance σ_k^2 and compare it to the threshold ϵ. When there is no error, all sub-ensembles should have a variance below the threshold. If a failure occurs, for example, on sensor j, then only one sub-ensemble should pass the test: $\mathcal{E}_{\tilde{l}_j}$, which consists of estimators that do not use sensor j. The appropriate threshold ϵ is chosen empirically.

To detect and handle multiple sensor failures, we repeat the following three steps:

1. **Ensemble Prediction**: Predict the pose using each model from the ensemble $\mathcal{E}$.
2. **Group Predictions**: Group the predictions by sensors not used to create the sub-ensembles $\mathcal{E}_l \subset \mathcal{E}$.
3. **Failure Detection**: Check the variance of each sub-ensemble to detect failure.

We start by masking $m = 2$ inputs to detect and handle one failure and increment that number as needed. To detect $n_{failures}$ failures, we build sub-ensembles $\mathcal{E}_l$ with $m = n_{failures}$ sensors masked. To identify which sensor(s) failed, we mask one additional sensor. We create the subsets $\mathcal{D}_i \subset \mathcal{D}$ for all possible $\binom{N}{m}$ mask configurations, training one model for each subset. This results in an ensemble $\mathcal{E}$ of $n = \binom{N}{m}$ models.

We repeat this process with additional sensors masked until there are not enough sensors left for a reliable prediction. For our experimental platform, we found that relying on three (out of $N = 8$) sensors is sufficient to have an acceptable precision, allowing us to detect and handle up to four failures.

To detect and react up to four failures concretely, we cover the following cases:

- One failure: mask 2 sensors and train $\binom{8}{2} = 28$ polynomial models.
- Two failures: mask 3 sensors and train $\binom{8}{3} = 56$ polynomial models.
- Three failures: mask 4 sensors and train $\binom{8}{4} = 70$ polynomial models.
- Four failures: mask 5 sensors and train $\binom{8}{5} = 56$ polynomial models.

This results in a total of $\sum_{m=2}^{5} \binom{N}{m} = 210$ polynomial models to handle all possible cases. We begin with $m = 2$ masked sensors and the first 28 polynomial models (nominal case, no failure), then increment the number of masked sensors and use additional polynomial models as needed. The failure detection and handling process remains the same at each stage: the three steps presented above.

5.1.3 Experimental Setup and Results

The objective of this section is to evaluate the performance of the proposed approach in terms of accuracy and speed, and to investigate its robustness against one or more sensor failures.

Experimental Setup

We use the *David* elastic neck Section 2.3.1 for the experiments, as shown in Fig. 2.9. The platform is equipped with a marker target from an external camera tracking system that provides ground truth data for the pose. The tracking system is used only for data collection and analysis and is not needed afterwards. The pose dependent length information is retrieved at a frequency of 300Hz. By commanding different tendon tensions, the neck achieves different poses.

Data-Driven Pose Estimation

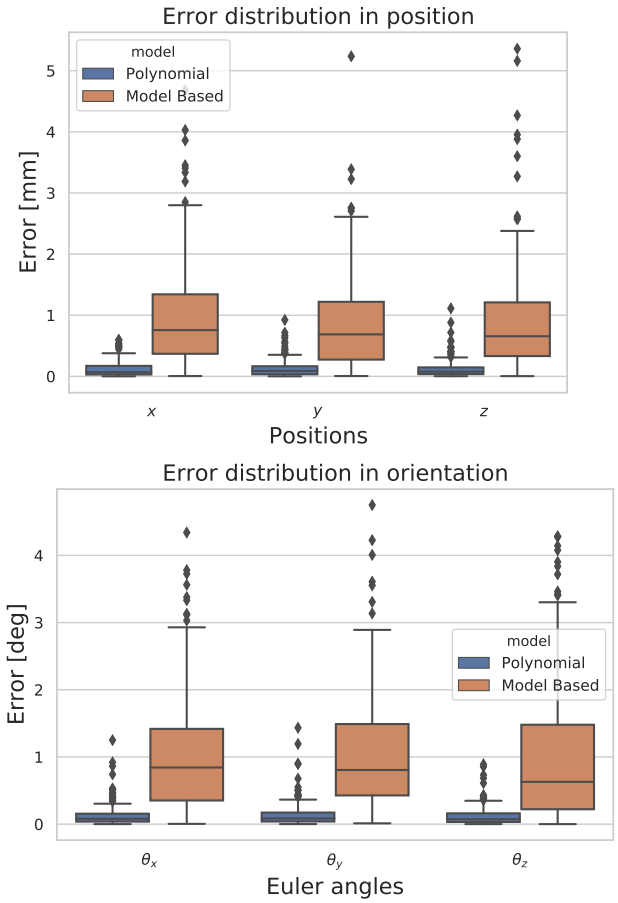


Figure 5.4: Error distribution in position and in orientation on 180 static poses for the polynomial model (data-driven) and the model-based approach [DCR+19].

Static Pose Estimation. To evaluate the performance of our six-DoF estimator, we command the elastic neck to achieve 200 static poses. For each pose, we obtain the ground

truth $\boldsymbol{q}$ using an external camera tracking system [Met16] and calculate the prediction error in both position and orientation. To train the models, we randomly sample 20 poses and use the remaining data points for testing. Each model is trained with the same 20 data points, but with different sensors masked, resulting in each model using a distinct set of sensors as input. This process is repeated ten times with different random seeds. To estimate the runtime of each method, we perform 1000 predictions and average the time taken. This test is performed on a computer equipped with 8 *Intel i7-8550U CPUs* clocked at 1.80 GHz.

Dynamic Pose Estimation. We then evaluate the pose estimation method, trained only on static poses, during dynamic motion. For this evaluation, we record the ground truth and the estimated pose along a predefined trajectory.

Results

Static Pose Estimation. The results are summarized in terms of mean error and standard deviation over the test poses in Table 5.1, with runtimes included when available. The error distribution in position and orientation is shown in Fig. 5.4.

Overall, the data-driven approaches demonstrate fast execution speeds (about 5000 Hz) and higher accuracy than the model-based approach: the mean error is reduced by up to five times (cf. Table 5.2). As expected, the linear model runs faster but with a slight loss in accuracy compared to the polynomial model.
It is worth noting that, although the model-based approach appears less accurate in this comparison, it remains relatively accurate and fast compared to other model-based pose estimation techniques as discussed in [DCR+19].

	Model-Based [DCR+19]	Linear	Polynomial
Runtime (ms)	N/A +/- N/A	**0.1 +/- 0.0**	0.2 +/- 0.0
	Static Pose Estimation		
Error in position (mm)	1.1 +/- 1.0	0.3 +/- 0.3	**0.2 +/- 0.2**
Error in orientation (deg)	0.9 +/- 0.8	0.3 +/- 0.3	**0.1 +/- 0.1**
	Dynamic Pose Estimation		
Error in position (mm)	5.1 +/- 2.3	1.7 +/- 1.4	**1.6 +/- 1.4**
Error in orientation (deg)	3.8 +/- 3.0	0.8 +/- 0.6	**0.5 +/- 0.4**

Table 5.1: Comparison of mean runtime and error (in both position and orientation) for each method. The data-driven approaches are both faster and more accurate than the model-based approach. For each metric, the best mean value is highlighted. "N/A" indicates that the data is not available.

Dynamic Pose Estimation. To evaluate the performance for dynamic pose estimation, we drive the experimental platform through a sequence of static poses and record the estimated pose for both the model-based and our approach. The mean error along the trajectory of 200 poses is presented in Table 5.1, and the corresponding trajectories of four estimated coordinates (x, θ_x, θ_y, θ_z) are shown in Fig. 5.5.

Similar to static poses, the data-driven approaches perform best, with the polynomial

model being more accurate than the linear model. The mean error reported in Table 5.1 is higher for dynamic poses than for static poses for two reasons. First, it includes the transition error between static poses, which is not accounted for during training. Second, when considering trajectories, static errors at fixed poses accumulate, resulting in a higher mean value.
It is worth noting that these models were trained on static poses only, and including dynamic poses in the training set could potentially improve the results.

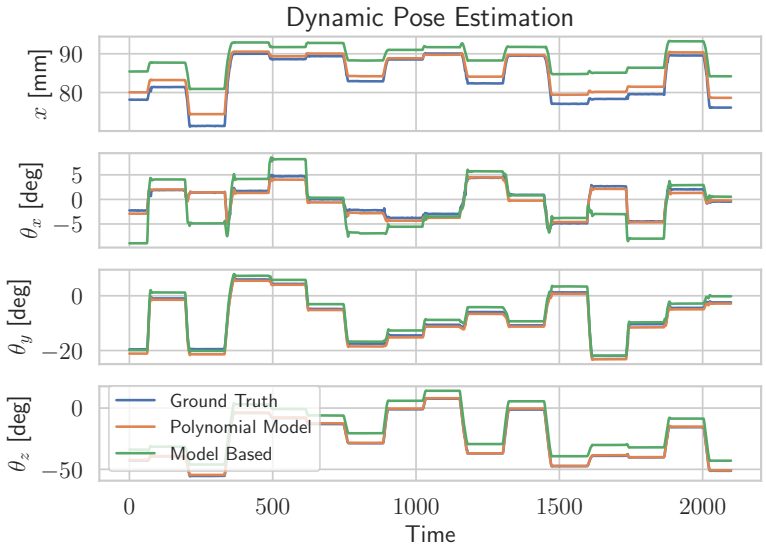


Figure 5.5: Qualitative evaluation of the polynomial estimators (data-driven) and model-based for dynamic pose estimation.

Hyperparameters study

To investigate the impact of different hyperparameters (model type, training set size, and number of sensors) on performance, we compare the baseline regressors (trained with 20 datapoints and 8 sensors) to several variants. The results of this comparison are presented in Table 5.2.

Effect of the training set size. We investigate the impact of varying the training set size from five datapoints to 40 datapoints on the performance of the polynomial model. Our results indicate that the performance improves with more training data, but only up to a certain point (30 datapoints in this case). Beyond this point, adding more data does not lead to significant improvements, likely due to the presence of irreducible error (sensor noise) and the limited capacity of the polynomial model. Although the results with 40 datapoints are slightly worse, the difference is not statistically significant.

Effect of the type of model. We compare linear, polynomial, and neural network[1] models. While the neural network achieves good performance, it requires more training data to reach the same level of accuracy as the other two models. In addition, the neural network requires hyperparameter tuning (mini-batch size, learning rate, etc.) and has a significantly larger number of parameters (approximately 3×10^6 compared to 276 for

[1]The neural network consists of 2 fully connected layers of 256 units each, trained to convergence with the Adam optimizer using a learning rate of 1×10^{-3} and a batch size of 20 (the size of the training set).

	Position error (mm)	Orientation error (deg)
Model-Based	1.1 +/- 1.0	0.9 +/- 0.8
Linear baseline	0.3 +/- 0.3	0.3 +/- 0.3
Polynomial baseline	0.2 +/- 0.2	**0.1 +/- 0.1**
Neural network (2 layers)	0.8 +/- 0.6	0.8 +/- 0.7
Linear (3 sensors)	1.0 +/- 0.7	1.2 +/- 1.0
Linear (4 sensors)	0.4 +/- 0.3	0.5 +/- 0.6
Polynomial (3 sensors)	0.9 +/- 0.6	1.2 +/- 0.9
Polynomial (4 sensors)	**0.1 +/- 0.1**	**0.1 +/- 0.1**
Polynomial (5 datapoints)	0.9 +/- 0.9	1.1 +/- 0.9
Polynomial (10 datapoints)	0.5 +/- 0.5	0.6 +/- 0.5
Polynomial (40 datapoints)	0.2 +/- 0.3	0.3 +/- 0.2

Table 5.2: Ablation study: influence of the amount of training data, number of sensors, and type of model on performance. Results with the lowest mean error are highlighted. Baseline models are trained using 20 datapoints and 8 sensors.

the polynomial model). Therefore, a second-order polynomial model is sufficient for our purposes.

Effect of the number of sensors. We compare the performance of the baseline linear and polynomial models (using 8 sensors) to models with fewer sensors as inputs (3 and 4 sensors). As expected, adding more sensors reduces the error for the linear model. For the polynomial model, there is almost no change with more than four sensors; the results are slightly worse, but the difference is not significant. As discussed in Section 5.1.2, having redundant sensor information is crucial for detecting and handling potential failures. Therefore, using 8 sensors is preferable because it allows for the detection of more failures.

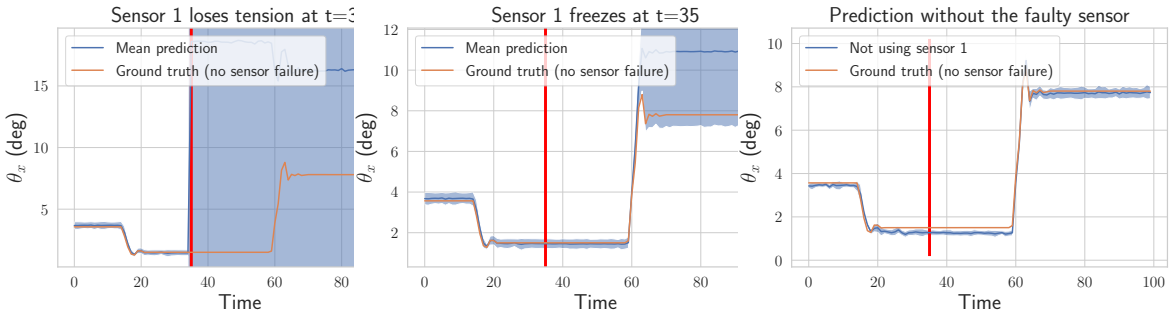
Effect of the placement of sensors. Since the polynomial model performs well with only four sensors, the placement of the four additional sensors has little impact on accuracy. However, the placement of the sensors does affect the ability to detect failures: if a sensor's information is not useful for prediction, the polynomial model will assign a weight close to zero to that input feature. As a result, if such a sensor fails, the failure will not be detected, but the pose will still be accurately estimated.

Failure Detection and Handling

To assess the effectiveness of our approach in detecting and handling failures, we simulate sensor losses while the neck was in motion. We consider two types of failures: (1) a length sensor outputting incorrect values due to a tendon becoming slack, resulting in zero outputs, and (2) a sensor freezing and outputting a constant value.

In Figs. 5.6a and 5.6b, we show the effect of both failures on the uncertainty: a jump is observed immediately after the loss of tension in Fig. 5.6a. When the sensor freezes (cf. Fig. 5.6b), the failure is detected only when the neck position changes. In both cases, the variance increases significantly, so no careful tuning of the detection threshold is required. To demonstrate that the method can handle multiple failures, we simulate the loss of four sensors in Fig. 5.7.

In Figs. 5.6c and 5.7, we display the prediction over time for the sub-ensemble without the faulty sensors, using the ground-truth pose as reference. The approach successfully



(a) Tendon associated with sensor $\tilde{l}_1$ goes slack

(b) Sensor $\tilde{l}_1$ freezes

(c) Prediction with the sub-ensemble $\mathcal{E}_{\tilde{l}_1}$ that does not use the faulty sensor

Figure 5.6: Example of failure detection and handling on a recorded trajectory. We simulate two types of failures indicated by the red vertical line. These failures occur at time $t = 35$: (a) One tendon becomes slack, causing the associated sensor $\tilde{l}_1$ to output zeros (b) The sensor $\tilde{l}_1$ freezes, causing it to output a constant value. For (a) and (b), we plot the mean prediction (dark blue) from the ensemble of estimators along with the uncertainty (shaded blue area). In both scenarios, the uncertainty exceeds the threshold, allowing the method to detect the failures and select the sub-ensemble that does not use the faulty sensor. In (c), we plot the prediction of this sub-ensemble, along with the corresponding ground truth. The method accurately predicts the pose even with a faulty sensor.

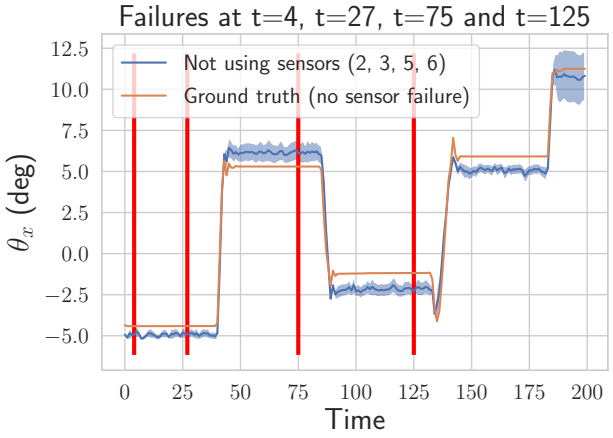


Figure 5.7: We simulate four failures along a recorded trajectory, indicated by the red vertical lines. The proposed method automatically detects these failures at the following times: sensor $\tilde{l}_6$ at t=4, $\tilde{l}_2$ at t=27, $\tilde{l}_5$ at t=75, and $\tilde{l}_3$ at t=125. It then selects a sub-ensemble of estimators that do not use the faulty sensors and continues to predict the pose accurately. The plot shows the pose prediction along with the mean and variance for the selected sub-ensemble of estimators.

detects all the failures and handles them by using the sub-ensemble not affected by the sensor losses. As a result, the proposed method can robustly and accurately (cf. Table 5.2) predict the pose even with multiple sensor failures.

5.1.4 Discussion

Data-Driven Approaches for Pose Estimation. In the previous sections, we have shown that data-driven approaches are viable alternatives for estimating the pose of a tendon-driven continuum mechanism. The linear and polynomial models have several advantages, including fast training times, minimal data requirements, and improved accuracy over the model-based approach. Model-based methods rely on assumptions about sensor linearity and perfectly known kinematics. If these assumptions are violated, they result in larger pose estimation errors. In contrast, the polynomial model learns a direct mapping from sensor readings to measured pose without relying on these assumptions, resulting in improved accuracy.

Handling sensor failures with ensembling and domain knowledge. To detect and handle sensor failures, we leverage ensembling techniques and minimal domain knowledge. By clustering different models, our method accurately predicts the pose even in the presence of four out of eight faulty sensors.

The presented method provides a computationally fast and accurate pose estimation approach. This method was used in the previous chapter to train an RL controller (Chapter 4) and will be further exploited in the following section to improve training time and final performance.

5.2 Learning to Control an Elastic Neck

In the following section, we further investigate the task presented in Section 4.5.3. The goal is to control *David*'s elastic neck to follow trajectories and reach desired poses using the pose estimator described in the previous sections. Compared to Section 4.5.3 where learning was done from scratch and without prior knowledge, we explore here different ways of guiding the RL agent.

5.2.1 Goal Conditioned Reinforcement Learning

First, we reformulate the RL problem as a goal-conditioned RL task. The agent receives as input the estimated state of the robot $\mathbf{s}_t$ and the desired goal g_{desired}, a 4D pose it should reach. When collecting interactions with the environment, we also store the current achieved goal g_{achieved}. This allows to use the Hindsight Experience Replay (HER) [AWR+17] method, implemented in SB3 (see Chapter 3), to relabel transitions. In short, during training, HER samples alternative goals g' from the set of achieved goals and then recomputes the reward for those new desired goals. Thus, it creates virtual transitions that are more likely to contain successful experiences (reaching a desired pose in our case). Those virtual transitions are sampled together with real transitions.

In Fig. 5.8, we display the results when training on the real robot. We can see that this simple reformulation guides the RL agent which needs only half the time to learn a controller (compared to Section 4.5.3). The goal-conditioned formulation allows the agent to learn from failed attempts, improving the data-efficiency of the training.

5.2.2 Combining Learned Feedforward and Feedback Controller

To further improve the sample efficiency and performance of the controller, we investigate how to guide the RL agent with domain knowledge. To this end, we re-use the pose estimator presented in the previous sections.

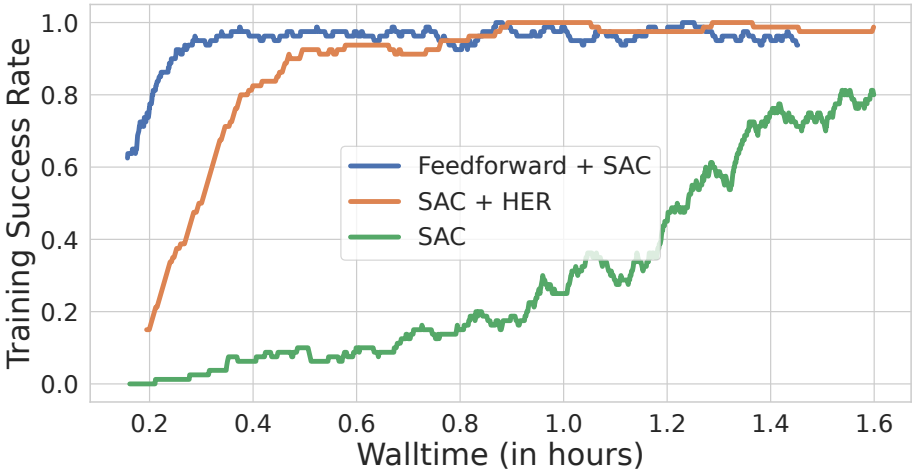


Figure 5.8: Training with Hindsight Experience Replay (HER) on the real robot, and combining feedforward and feedback control. SAC corresponds to the experiment presented in Section 4.5.3.

The linear pose estimator from Section 5.1.2 predicts the neck pose given the tendons lengths: $f_\Theta(\tilde{\boldsymbol{l}}) = \Theta^T \tilde{\boldsymbol{l}} = \hat{\boldsymbol{q}}$. If we invert the mapping[2], we obtain a feedforward controller: the estimator $\hat{\boldsymbol{l}} = f_\Theta^{-1}(\boldsymbol{q})$ predicts desired tendons lengths $\hat{\boldsymbol{l}}$ for a given desired pose $\boldsymbol{q}$.

The feedforward already allows reaching some poses without any additional training, as we re-use the data acquired to learn the pose estimator. We then close the loop with reinforcement learning to refine the controller and improve the success rate. The RL controller only outputs a delta on top of the predicted feedforward action.

Fig. 5.8 shows the results on the robot. Because it leverages the feedforward controller, the RL agent learns very quickly to achieve the task and reaches a success rate above 90% after only 15 minutes. This is eight time faster than in Chapter 4.

5.3 Conclusion

In this chapter, we presented a data-driven fault-tolerant approach to accurately predict the pose of a soft robot. Although model-free, the estimator outperforms its model-based counterpart and can be easily inverted to obtain a feedforward controller.

After learning from scratch in Chapter 4, we also show how to guide the RL agent with additional knowledge. Reformulating the problem to explicitly include the desired goal improves the sample efficiency, allowing the agent to learn from its mistakes using virtual transitions.

The best result, both in terms of performance and training time, is obtained by combining the learned feedforward model with RL. In this case, the agent only needs to learn a delta command on top of the feedforward controller that partially solves the task.

[2]Either using pseudo-inverse or training a new model on the inverse problem.

CHAPTER 6

Combining Oscillators and RL to Efficiently Learn Locomotion

In Chapter 4, we showed the use of reinforcement learning to teach a real quadruped robot to walk from scratch. However, this approach requires several hours of training and engineering effort, and the resulting gait may not be natural.

As in Chapter 5, where we combined domain knowledge with RL to accelerate robot training, we now explore how prior knowledge can be used to guide the agent and learn better locomotion controllers more quickly (tackling challenges ii. (Sample Efficiency) and iv. (Computational Resource Constraints)). Specifically, we propose a simple open-loop baseline controller that can solve locomotion tasks and compare it with DRL algorithms. We then build on the open-loop oscillators and close the loop with RL to improve performance.

Our focus in this chapter will be on the elastic quadruped presented in Section 2.3.2. By learning directly on the real hardware, we can take advantage of its springs. We will close this chapter with an experiment where learning on the real robot in a matter of minutes was essential for a mission with the International Space Station (ISS), where astronauts from the ISS operated the robot in a planetary exploration scenario.

6.1 An Open-Loop Baseline for RL Locomotion Tasks

The increasing complexity of Deep RL algorithms has led to a reproducibility crisis, with many implementation details required to achieve satisfactory performance [HDR+22, HIB+18a]. Moreover, even state-of-the-art methods struggle to solve simple problems, such as the Mountain Car environment [CSO18] and the Swimmer task [FLO+22, HGF+24]. This has led to a reassessment of the field, with some work advocating simpler, more scalable alternatives [RLTK17, SHC+17, MGR18]. We argue that the use of prior knowledge can reduce complexity in both algorithms and task formulation, especially for specific problem categories such as locomotion tasks.

To illustrate this, we introduce a minimal open-loop model-free strategy as a baseline for locomotion challenges. By comparing this baseline to DRL algorithms in various scenarios, our goal is not to replace them, but to highlight their limitations, provide insights, and encourage reflection on the costs of complexity and generality.

The following sections present the concrete contributions of this work:

- a simple open-loop model-free baseline for learning locomotion that can handle sparse rewards and high sensory noise with minimal parameters (on the order of tens, Section 6.1.2),

- demonstrating the importance of prior knowledge and policy structure in locomotion tasks (Section 6.1.3),

- an investigation of the robustness of RL algorithms to noise and sensor failure (Section 6.1.3),

- successful simulation-to-reality transfer without randomization or reward engineering, where DRL algorithms fail (Section 6.1.3).

6.1.1 Related Work

The quest for simpler RL baselines. Amid the trend of increasingly complex RL algorithms, some research has focused on developing simple yet effective baselines for robotic tasks. In particular, [RLTK17] explored the use of policies with simple parameterizations, highlighting the lack of robustness of RL agents. Concurrently, [SHC+17] investigated evolution strategies as an alternative to RL, taking advantage of their fast execution time to scale policy search. Augmented Random Search (ARS) [MGR18] is a more recent example of a population-based algorithm that optimizes linear policies.

Periodic policies for locomotion. Given the fundamental role of rhythmic movements in locomotion [Del80, CW80, Ijs08], oscillators have been employed in robot control to solve locomotion tasks [CI08, IAST13], with recent work focusing on quadruped robots [KS04, TZC+18, ICT+18, YZC+22, BI22, RSK+22]. Surprisingly, however, we found no prior studies exploring the use of open-loop oscillators in RL locomotion benchmarks.

Reinforcement learning for locomotion. Obtaining locomotion controllers is commonly based on fast simulators and massive parallelism [MLH+22, RHRH22]. It usually consists of learning the controllers in simulation and subsequently deploying them on real hardware [LHW+20, KFPM21]. Learning in simulation, however, requires an accurate model of the robot [HLD+19], complex reward engineering, and there is no guarantee that a controller working in simulation can be deployed on the real robot. Heavy randomization and generation of diverse training environments help mitigate these issues [FKMP21, RHRH22], but still requires engineering efforts to find the right balance between environments too hard to solve and ones that do not transfer to the real robot. In the first part of this chapter, we explore how to reduce the engineering effort by using prior knowledge and focus on learning directly on the real hardware in the second part.

Training on real robot. Thanks to more robust hardware [BGC+22] and more sample-efficient algorithms [CVS+19, SKL23, WEH+22, HIH+22], learning directly on the real hardware in only a few hours is now possible [CSPD16, RKS21]. Despite these advances, training on real robots can nevertheless be time-consuming and often does not yield natural-looking gaits [SKL23, WEH+22]. In the present chapter, we aim both at reducing the training time (and thus the robot wear and tear) and obtaining gaits that look natural by 1) providing domain knowledge in the form of open-loop oscillators and 2) exploiting the natural dynamics of the robot, in particular its elastic joints.

6.1.2 Open-Loop Oscillators for Locomotion

Inspired by nature and central pattern generators, as explored by [RBI06, RSK+22, BI22], we propose using nonlinear oscillators with phase-dependent frequencies to generate desired motions for each actuator. The equation governing one oscillator is:

$$\begin{aligned} q_i^{\text{des}}(t) &= a_i \cdot \sin(\psi_i(t) + \varphi_i) + b_i \\ \dot{\psi}_i(t) &= \begin{cases} \omega_{\text{swing}} & \text{if } \sin(\psi_i(t) + \varphi_i) > 0 \\ \omega_{\text{stance}} & \text{otherwise} \end{cases} \end{aligned} \tag{6.1}$$

where q_i^{des} is the desired position for the i-th joint, a_i, ψ_i, φ_i and b_i are the amplitude, phase, phase shift and offset of oscillator i. ω_{swing} and ω_{stance} control the oscillations in rad/s for the swing and stance phases, respectively. To reduce the search space, we employ the same frequencies ω_{swing} and ω_{stance} for all actuators.

This formulation is both computationally efficient and simple to implement; since we do not have any feedback terms, all desired positions can be pre-computed. The phase shift φ_i serves as a coupling term, allowing joints with the same phase shift to oscillate synchronously. However, unlike previous work, the phase shift is learned rather than pre-defined.

We optimize the oscillator parameters using black-box optimization (BBO), namely the CMA-ES algorithm [HMK03, Han09] implemented within the Optuna library [ASY+19]. Furthermore, BBO's reliance on episodic returns rather than immediate rewards makes the baseline robust to delayed or sparse rewards. Finally, a Proportional-Derivative (PD) controller converts the desired joint positions generated by the oscillators into desired torques.

6.1.3 Results

We benchmark DRL algorithms against our baseline by conducting experiments on locomotion tasks, which encompass both simulated scenarios and real-world transfer to an elastic quadruped robot. Our study is guided by three main research questions:

- How do simple open-loop oscillators compare to DRL methods in terms of performance, computational efficiency, and parameter efficiency?
- What is the relative robustness of RL policies versus the open-loop baseline to sensor noise, failures, and external disturbances?
- How effectively do learned policies transfer to a real-world robot when trained without randomization or reward engineering?

By addressing these questions, we aim to provide a comprehensive understanding of the strengths and weaknesses of our proposed baseline and highlight the potential benefits of incorporating prior knowledge in robotic control.

Results on the MuJoCo locomotion tasks

We evaluate the performance of our method on five MuJoCo v4 locomotion tasks (Ant, HalfCheetah, Hopper, Walker2d, Swimmer) included in the Gymnasium v0.29.1 library [TTK+23]. We compare our approach to three established DRL algorithms: Deep

Deterministic Policy Gradients (DDPG), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC). To ensure a fair comparison, we adopt the hyperparameter settings from the original papers, except for the swimmer task, where we fine-tune the discount factor ($\gamma = 0.9999$) according to [FLO+22]. Additionally, we also benchmark Augmented Random Search (ARS), a population-based algorithm that uses linear policies.

Our choice of baselines features one representative example per algorithm category: DDPG as a historical algorithm (see Section 2.2), PPO for on-policy, SAC for off-policy, and ARS for population-based methods and simple model-free baselines. We select SAC [HHZ+18] due to its strong performance in continuous control tasks [HGF+24] and its shared components with newer variants. SAC and its variants, such as TQC [KSGV20], REDQ [CWZR21], or DroQ [HIH+22], are frequently used in the robotics community [RSK+22, SKL23]. For the reward functions, we keep the ones provided by Gymnasium, except for ARS, where we remove the alive bonus to match the results from the original paper.

The RL agents are trained for one million steps. To obtain quantitative results, we run each experiment 10 times with distinct random seeds. We follow the recommendations by [ASC+21] and report performance profiles, probability of improvements in Fig. 6.1, and aggregated metrics with 95% confidence intervals in Fig. 6.2. The scores are normalized over all environments using a random policy for the minimum and the maximum performance of the open-loop oscillators.

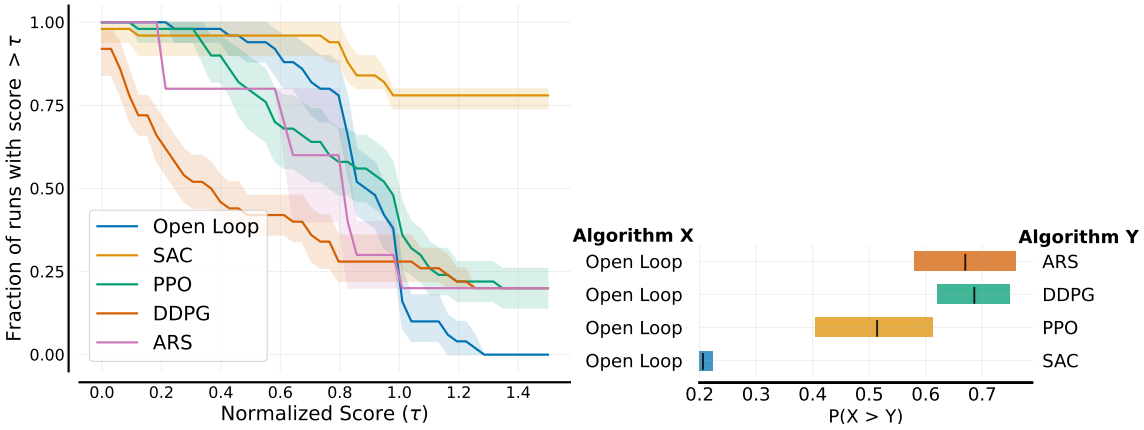


Figure 6.1: Performance profiles on the MuJoCo locomotion tasks (left) and probability of improvements of the open-loop approach over baselines, with a 95% confidence interval.

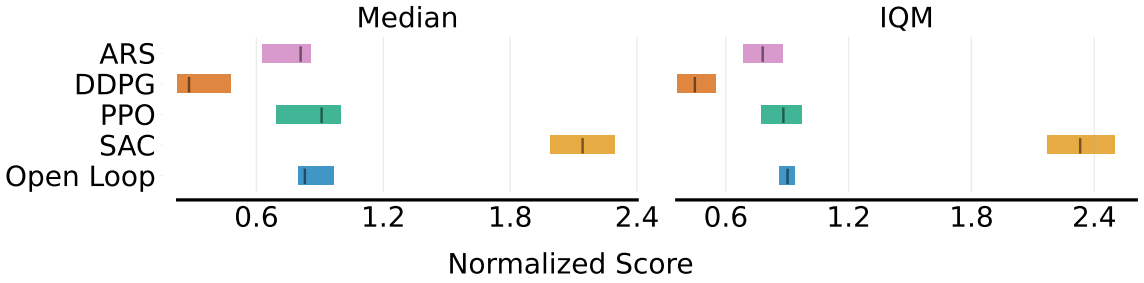


Figure 6.2: Metrics results on MuJoCo locomotion tasks using median and interquartile mean (IQM), with a 95% confidence interval.

Performance. As displayed in Figs. 6.1 and 6.2, the open-loop oscillators show respectable performance across all five tasks, despite their minimalist design. They outperform ARS and DDPG, a simple baseline and a classic DRL algorithm, and perform comparably to PPO. In particular, this is achieved with only a dozen parameters, as opposed to the thousands typically required by DRL algorithms. Our results suggest that simple oscillators can effectively compete with complex RL methods for locomotion, and do so in an open-loop fashion. However, the open-loop approach has its limitations, as the baseline does not reach the maximum performance of SAC.

Table 6.1: Runtime comparison to train a policy on HalfCheetah, one million steps using a single environment, no parallelization.

	SAC		PPO		DDPG		ARS		Open-Loop	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Runtime (in min.)	80	30	10	14	60	25	5	N/A	**2**	N/A

Runtime. A comparison of the runtime of the different algorithms[1], as presented in Table 6.1, highlights the benefits of simplicity over complexity. Remarkably, ARS requires just five minutes of CPU time to train on a single environment for one million steps, whereas open-loop oscillators are twice as fast. This efficiency is especially advantageous when deploying policies on embedded platforms with limited computing resources[2]. Additionally, both methods can be easily scaled using asynchronous parallelization to further reduce training time. On the other hand, more complex methods like SAC need a GPU to achieve reasonable runtimes (15 times slower than open-loop oscillators), even with the help of JIT compilation.

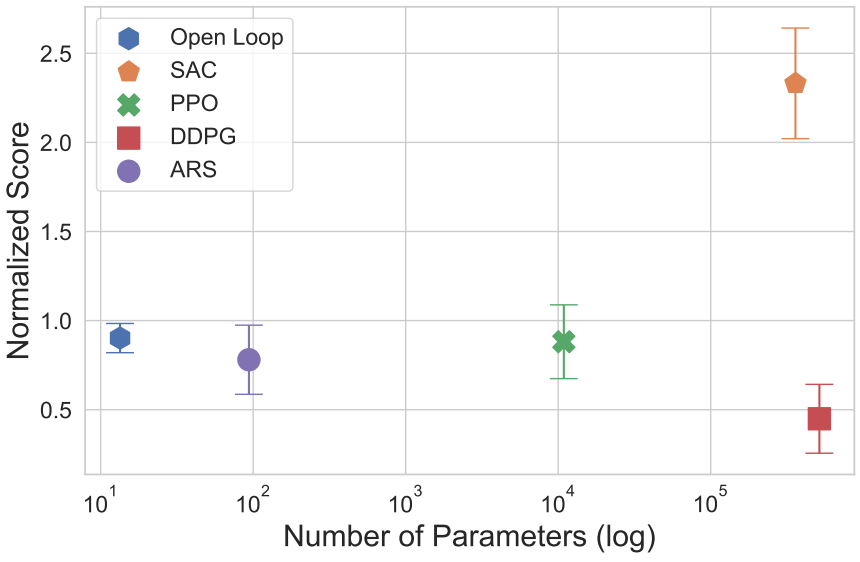


Figure 6.3: Parameter efficiency of the different algorithms. Results are presented with a 95% confidence interval and score are normalized with respect to the open-loop baseline.

[1] We display the runtime for HalfCheetah only, the computation time for the other tasks is similar.
[2] A concrete example will be presented at the end of this chapter.

Parameter efficiency. As shown in Fig. 6.3, the open-loop oscillators stand out for their simplicity and performance with respect to the number of optimized parameters. On average, the proposed approach has 7x fewer parameters than ARS, 800x fewer than PPO, and 27,000x fewer than SAC. This comparison underscores the importance of choosing an appropriate policy structure that delivers satisfactory performance while minimizing complexity.

Robustness to sensor noise and failures

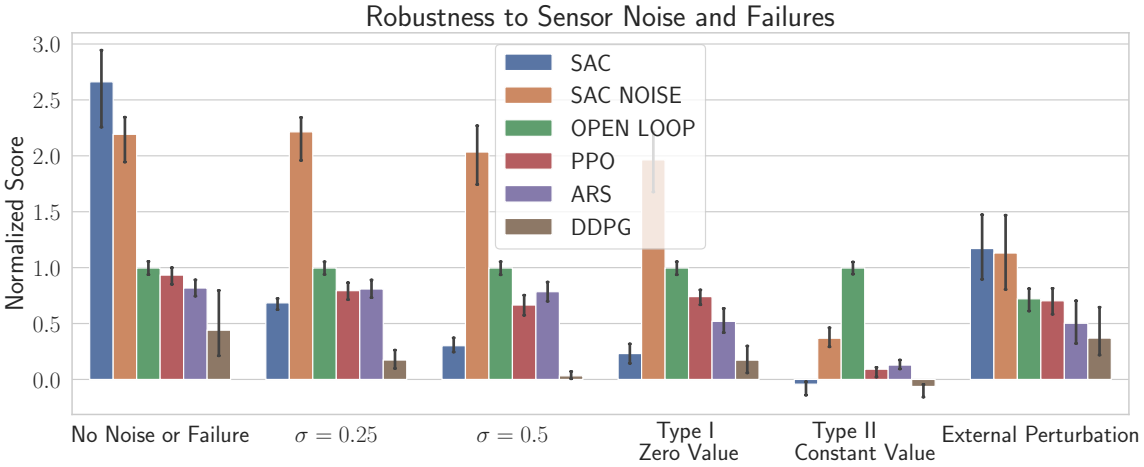


Figure 6.4: Robustness to sensor noise (with varying intensities), failures of Type I (all zeros) and II (constant large value) and external disturbances. All results are presented with a 95% confidence interval and score are normalized with respect to the open-loop baseline.

In this section, we evaluate the robustness of the trained agents from the previous section against sensor noise, failures, and external disturbances [DALM+20, SGS+21]. To investigate the impact of noisy sensors, we introduce Gaussian noise with varying intensities into one sensor signal (specifically, the first index in the observation vector, which provides the end-effector position). We simulate two types of sensor malfunctions to study the robustness against sensor faults: Type I failure involves outputting zero values for one sensor, while Type II failure generates a constant value with a larger magnitude (set to five in our experiments). Finally, we assess the robustness to external disturbances by applying perturbations with a force of 5N in randomly chosen directions with a probability of 5% (around 50 impulses per episode). By examining the agents' performance under these scenarios, we can evaluate their ability to adapt to imperfect sensory input and react to perturbations. We also study the effect of randomization by training SAC with a Gaussian noise with intensity $\sigma = 0.2$ on the first sensor (SAC NOISE in the figure).

In the absence of noise or failures, SAC outperforms simple oscillators on most tasks, with the exception of the Swimmer environment. However, as seen in Fig. 6.4, SAC's performance deteriorates quickly when exposed to noise or sensor faults. This is also the case for the other RL algorithms, where ARS and PPO are the most robust, but still show degraded performance. In contrast, open-loop oscillators remain unaffected, except when exposed to external disturbances, as they do not rely on sensors. This highlights one of the primary advantages and limitations of open-loop control.

The performance of SAC trained with noise on the first sensor (SAC NOISE) shows that it is possible to mitigate the effects of sensor noise. This finding, together with the performance of the open-loop controller, suggests that the first sensor is not essential for achieving good results in the MuJoCo locomotion tasks. SAC with randomization on the first sensor has learned to ignore its input, while SAC without randomization shows a high sensitivity to the value of this uninformative sensor. This illustrates a limitation of DRL algorithms, which can be sensitive to useless inputs.

Simulation to Reality Transfer on an Elastic Quadruped

Figure 6.5: Robotic quadruped with elastic actuators in simulation (left) and real hardware (right)

The open-loop approach presents a promising baseline for locomotion control on real robots, given its computational efficiency, robustness to sensor noise, and satisfactory performance. To assess its potential for real-world applications, we investigate the transferability of the simulation results to the *bert* quadruped robot equipped with serial elastic actuators (introduced in Section 2.3.2).

We employ a simulation of the robot in PyBullet [CB21] to perform our evaluation. The *bert* simulation incorporates a model of the elastic joints but omits motor dynamics. The objective is to achieve maximum forward speed: we define the reward as displacement along the desired axis and limit each episode to five seconds of interaction. The agent receives the current joint positions q and speeds $\dot{q}$ as observations and commands the desired joint positions q^{des} at a frequency of 60Hz.

In this experiment, we compare the open-loop approach with the best performing algorithm from Section 6.1.3, which is SAC. Both algorithms are given a budget of one million steps for training. In particular, we do not use any randomization or task-specific techniques during the training process. Our aim is to understand the strengths and weaknesses of RL relative to the open-loop baseline in a simple simulation-to-reality setting. We test the policy learned in simulation on the real robot for ten episodes.

As presented in Table 6.2, SAC outperforms the open-loop oscillators in simulation, achieving a mean speed of 0.81 m/s compared to 0.55 m/s over ten runs, similar to the results in Section 6.1.3. However, a closer examination reveals that the policy learned by SAC generates high-frequency commands, making it unlikely to transfer to the real robot – a common challenge faced by RL algorithms [RKS21, BI22][3]. When deployed on the real robot, the jerky motion patterns result in suboptimal performance (0.04 m/s), potentially damaging commands for the motors, and increased wear-and-tear.

[3]Chapter 4 discusses this issue in more detail.

Table 6.2: Results of simulation-to-reality transfer for the elastic quadruped locomotion task. We report mean speed and standard error over ten test episodes. SAC performs well in simulation, but fails to transfer to the real world.

	SAC		Open-Loop	
	Sim	Real	Sim	Real
Mean speed (m/s)	**0.81** +/ 0.02	0.04 +/ 0.01	0.55 +/ 0.03	**0.36** +/ 0.01

In contrast, our open-loop oscillators, with less than 25 adjustable parameters, produce smooth outputs and achieve good performance on the real robot. Although there is still a gap between simulation and reality, it is much smaller than for the RL algorithm.

6.1.4 Ablation Study

In this section, we analyze how design choices from Eq. (6.1) affect performance. In particular, we explore the impact of using phase-dependent frequencies (where we set $\omega_{\text{swing}} = \omega_{\text{stance}} = \omega$) and the role of phase shifts φ_i between oscillators (where we set $\varphi_i = 0$). The results are presented in Figs. 6.6 and 6.7 and Table 6.3.

The equations of the different variants of Eq. (6.1) are:

$$\begin{aligned} q_i^{\text{des}}(t) &= a_i \cdot \sin(\omega \cdot t + \varphi_i) + b_i && \text{No } \omega_{\text{swing}} \\ q_i^{\text{des}}(t) &= a_i \cdot \sin(\psi_i(t)) + b_i && \text{No } \varphi_i \\ q_i^{\text{des}}(t) &= a_i \cdot \sin(\omega \cdot t) + b_i && \text{No } \varphi_i \text{ No } \omega_{\text{swing}} \end{aligned} \tag{6.2}$$

where $\psi_i(t)$ is the same as in Eq. (6.1).

For the HalfCheetah, Swimmer, and Ant tasks, a single frequency ω proves sufficient. However, for the Hopper environment, it is crucial to have phase-dependent frequencies. Phase shifts φ_i are necessary when joints cannot move synchronously, as observed in the Swimmer task. For quadrupeds, these phase shifts φ_i are essential for representing gaits and capturing leg symmetries.

Table 6.3: Results on MuJoCo locomotion tasks (mean and standard error are displayed) with different variant of the approach.

	Open-Loop			
	No φ_i No ω_{swing}	No φ_i	No ω_{swing}	Full
Ant-v4	1167 +/- 3	1173 +/- 3	1239 +/- 8	1235 +/- 6
HalfCheetah-v4	2221 +/- 27	2245 +/- 30	2532 +/- 42	2400 +/- 31
Hopper-v4	929 +/- 9	785 +/- 28	986 +/- 7	1241 +/- 30
Swimmer-v4	-119 +/- 8	-82 +/- 6	356 +/- 0	356 +/- 0
Walker2d-v4	1484 +/- 36	1482 +/- 34	1140 +/- 32	1508 +/- 27

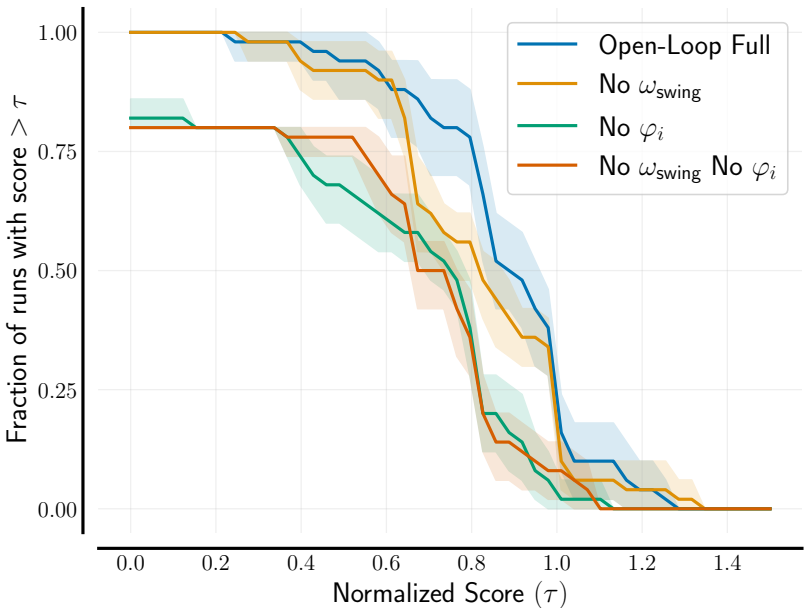


Figure 6.6: Performance profiles on the MuJoCo locomotion tasks using different variants of the open-loop approach, with a 95% confidence interval.

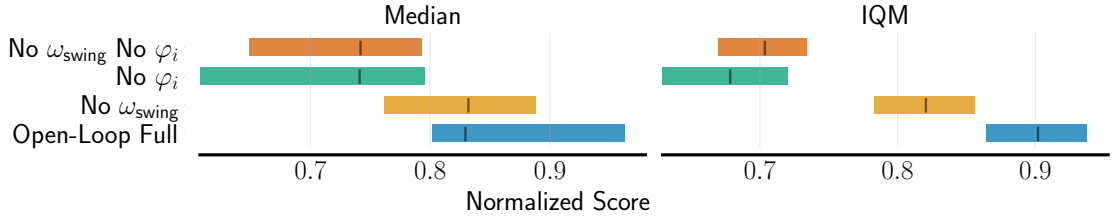


Figure 6.7: Metrics results on MuJoCo locomotion tasks for the different variants using median and interquartile mean (IQM), with a 95% confidence interval.

6.1.5 Discussion

A simple open-loop model-free baseline. We present a simple, open-loop model-free approach that achieves satisfactory performance on standard locomotion tasks without relying on complex models or expensive computational resources. Although it may not outperform DRL algorithms in simulation, this method offers several advantages for real-world applications, including efficient computation, ease of deployment on embedded systems, smooth control outputs, and robustness to sensor noise. These features help bridge the gap between simulation and reality, avoiding common problems associated with DRL algorithms, such as jerky motion patterns [RKS21] or convergence to a "bang-bang" controller [SGS+21]. Our approach is specifically designed for locomotion tasks, but its simplicity does not compromise its versatility, allowing it to be applied to a variety of locomotion tasks and transfer to a real robot with just a few adjustable parameters, while remaining model-free.

The cost of generality. Deep RL algorithms for continuous control often seek generality by using a versatile neural network architecture for the policy. However, this pursuit

of generality in the algorithm comes at the cost of specificity in the task design. Indeed, the reward function and action space must be carefully designed to solve the locomotion task and avoid solutions that exploit the simulator but do not transfer to real hardware. Our study and recent work [ICT+18, BI22, RSK+22] suggest integrating domain knowledge into policy design. Even minimal knowledge, such as simple oscillators, can reduce the search space and the need for complex algorithms or reward design.

Unexpected results. While the success of open-loop oscillators in the SWIMMER environment is expected, their effectiveness in the WALKER, HOPPER, or elastic quadruped environments is more surprising, as one might assume that feedback control or inverse kinematics would be needed to balance the robots or learn a meaningful open-loop policy. Although previous studies have already shown that periodic movements are at the heart of locomotion [Ijs08], we argue that the periodic motion required can be surprisingly simple. With the ARS algorithm, [MGR18] showed that simple linear policies can be used to solve locomotion tasks. Our work takes this a step further by reducing the number of parameters by a factor of ten and removing the state as an input.

Limitations Naturally, open-loop control alone is not a definitive solution to locomotion challenges. By design, open-loop control is sensitive to disturbances and cannot recover from a potential fall. In such cases, it becomes essential to close the loop with RL to adapt to changing conditions, maintain stability, or track a desired goal. A hybrid approach that leverages the strengths of feedforward (open-loop) and feedback (closed-loop) control offers an intermediate solution, as seen in various fields of engineering [GGS00, AM08, DSBG+17]. By combining the fast computation and noise resilience of open-loop control with the adaptability of closed-loop control, it enables reactive locomotion. This combination is explored in the next section.

6.2 Learning to Exploit Elastic Actuators: Combining Open-Loop Oscillators with RL

In this section, we investigate the combination of open-loop oscillators, presented in the previous section, with reinforcement learning. The idea is to exploit the strengths of the open-loop oscillators and learn only a delta on top of the feedforward term (similar to what we did in Section 5.2). We also reduce the dimensionality of the problem by generating trajectories in task space. This allows fast learning of different gaits directly on the elastic robot *bert* that take advantage of its springs.

6.2.1 Open-Loop Oscillators in Task Space

Similar to Section 6.1.2, we generate open-loop policies for locomotion using a system of nonlinear oscillators, that have phase-dependent frequencies. Compared to Section 6.1.2 where we were operating at the joint level, we reduce the search space and use one oscillator per leg, as done by previous work on CPGs [RBI06, RI08, APSI13].

As shown in Fig. 6.8, the output of each oscillator determines the foot trajectory in Cartesian space. The trajectory of each leg i depends on four parameters[4] that have to be tuned: ground clearance Δz_{clear} and penetration Δz_{pen}, step length Δx_{len}, swing ω_{swing} and stance ω_{stance} frequencies. We convert the output of one oscillator i to a desired foot

[4] In the remaining section, we fix manually the phase shift parameters φ_i to obtain desired known gaits (trot, pronk, ...). We lift this restriction in Section 6.4.1.

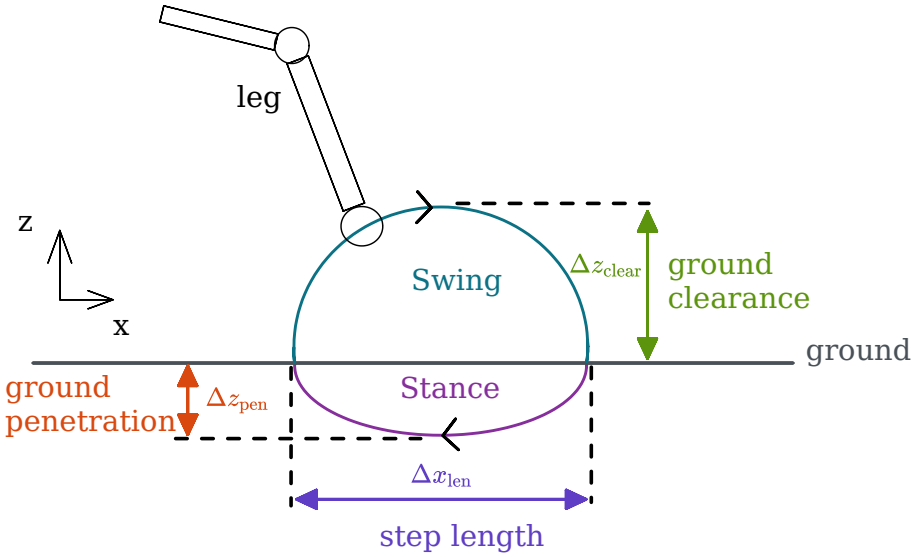


Figure 6.8: The parameters of the open-loop oscillators gait that are optimized.

trajectory following:

$$\begin{aligned} x_{\text{des},i} &= \Delta x_{\text{len}} \cdot \cos(\psi_i + \varphi_i) \\ z_{\text{des},i} &= \Delta z \cdot \sin(\psi_i + \varphi_i) \\ \Delta z &= \begin{cases} \Delta z_{\text{clear}} & \text{if } \sin(\psi_i + \varphi_i) > 0 \text{ (swing)} \\ \Delta z_{\text{pen}} & \text{otherwise (stance).} \end{cases} \end{aligned} \tag{6.3}$$

The most critical parameters for an elastically-actuated quadruped are the swing and stance duration, as they should match the natural dynamics of the robot [DSLBAS20, ASDS20].

Tuning these variables manually is time-consuming and usually results in suboptimal choices. Therefore, we automate their tuning using BBO (as in Section 6.1.2) and run the optimization directly on the real robot, avoiding any model mismatch.

6.2.2 Online Learning for Legged Locomotion

To learn a locomotion controller for the elastic legged robot, we combine two model-free methods. We first optimize the parameters of a selected gait to quickly obtain an open-loop controller that works in the nominal case without disturbances. Then, we refine the controller and make it more robust to perturbations by learning a RL controller that adds offsets to the oscillators output. The entire optimization process – depicted in Fig. 6.9 – is conducted directly on the real robot, removing the need for modeling, accurate simulators or simulation-to-reality transfer.

The oscillators provide a compact representation of limb end-effector trajectories (Eq. (6.3)) that can be adjusted via the set of parameters α shown in Fig. 6.8:

$$\alpha = \{ \Delta z_{\text{clear}}, \Delta z_{\text{pen}}, \Delta x_{\text{len}}, \omega_{\text{swing}}, \omega_{\text{stance}} \}. \tag{6.4}$$

While the BBO algorithm optimizes the parameters of the gait generated by the coupled oscillators (as in Section 6.1.2), we use RL to learn a reactive controller π as a set of corrective actions per leg i defined as $\boldsymbol{\pi}_i = [\pi_{x,i}(\mathbf{s}_t), \pi_{z,i}(\mathbf{s}_t)]$. The learned policy π adjusts

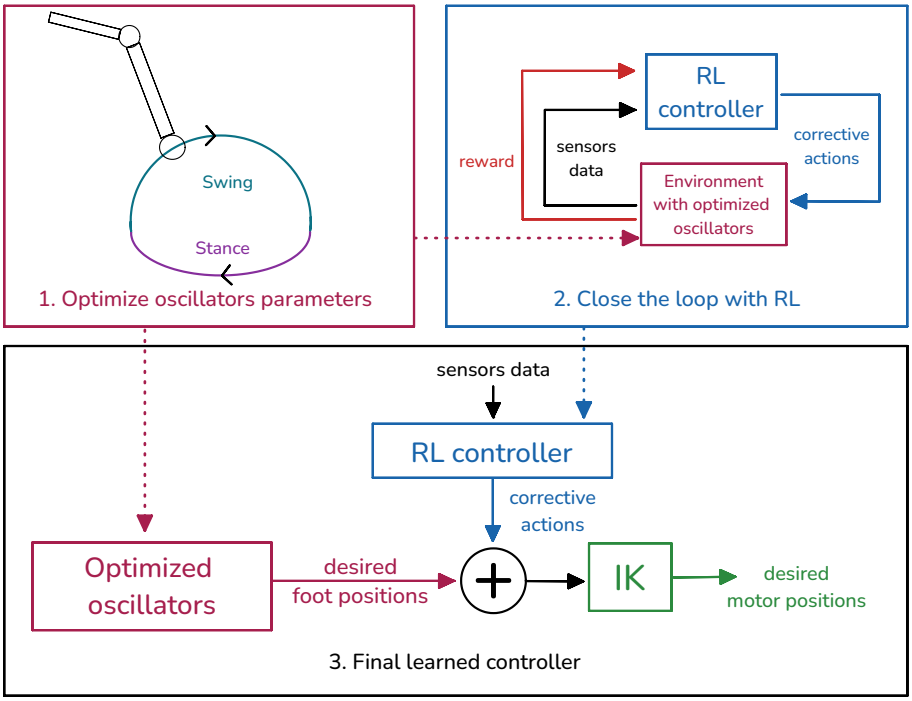


Figure 6.9: Overview of the proposed approach: oscillators parameters are first optimized using BBO, then a RL controller is trained on top and outputs corrective actions.

desired foot positions produced by the oscillators (Eq. (6.3)) given the current state of the robot $\mathbf{s}_t$, closing the control loop:

$$\begin{aligned} x_{\text{des},i} &= \Delta x_{\text{len}} \cdot \cos(\psi_i + \varphi_i) + \pi_{x,i}(\mathbf{s}_t) \\ z_{\text{des},i} &= \Delta z \cdot \sin(\psi_i + \varphi_i) + \pi_{z,i}(\mathbf{s}_t) \end{aligned} \tag{6.5}$$

6.2.3 Metrics for Elastic Robots

Objective functions of learning methods typically describe tasks (e. g. move forward as fast as possible), and do not enforce explicit usage of the compliant actuators. To measure if the learned controllers exploit the capability of the elastic robot, we define metrics that quantify the spring usage.

By storing and releasing energy, springs allow joints to reach much higher velocities than the motors [MKH21]. The peak joint velocity usually has a delay compared to the peak motor velocity. Therefore, we propose to compare the ratio between maximum joint and motor velocity over a trajectory for each joint:

$$\mathcal{R}_i^{\frac{\dot{q}}{\dot{\theta}}} = \frac{\max_t \dot{q}_i}{\max_t \dot{\theta}_i}. \tag{6.6}$$

Intuitively, if the springs are fully exploited, the ratio $\mathcal{R}^{\frac{\dot{q}}{\dot{\theta}}}$ should be greater than one (rigid robot baseline), meaning that the energy is stored and released at the right moment.

We also monitor the potential and kinetic energy to better understand how energy is transferred and how the learned controllers utilize the elastic actuators. We only consider

the kinetic energy related to the task (translational velocity of the center of mass) and therefore do not plot the rotational kinetic energy.

6.2.4 Task Specification

To evaluate the proposed approach, we focus on two dynamic gaits: trotting and pronking (jumping in place).

Trotting The first task we are interested in is to obtain a fast walking trot, where we optimize for mean forward speed ($f(\alpha) = -\frac{\Delta x}{\Delta t}$). We define the immediate reward as the distance traveled between two timesteps along the desired axis ($r_t = \dot{x}_{\text{robot}}$).

After optimization, we further evaluate the five best performing candidates for more episodes. This post-processing step is key to filtering out the evaluation noise, i. e. it helps to find parameters that lead to reliable gaits.

Pronking The second task is to jump in place (pronk) without falling or drifting from the starting position. In a pronking gait, all legs are synchronized: they all take off and land at the same time (as seen in Fig. 6.10). The step length is set to zero in that case. We penalize angular velocity for roll and yaw, distance to the starting point and reward velocity along the z axis (aligned with gravity field): $r_t = w_1|\dot{z}| - w_2(\dot{\psi}^2_{\text{roll}} + \dot{\psi}^2_{\text{yaw}}) - w_3\Delta^2_{xy}$, where $w_{1,2,3}$ are weights chosen in a way that the primary reward $|\dot{z}|$ has a higher magnitude than the secondary costs. We define the objective function for the BBO as the total reward per episode ($f(\alpha) = -\sum_t r_t$).

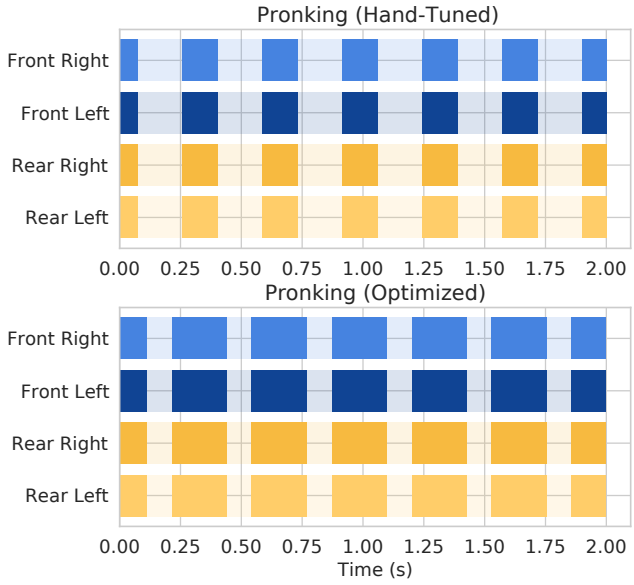


Figure 6.10: Pronking gaits for a 2-second period.

In all tasks, the RL agent receives as input $\mathbf{s}_t$ the current joint positions, velocities, torques, linear acceleration and angular velocity (from the IMU), and the desired foot position generated by the open-loop controller. The action space corresponds to offsets for each foot in Cartesian space.

6.2.5 Results

We evaluate our approach on two locomotion tasks (trotting and pronking) to answer the following questions:

- Can we quickly learn a controller directly on the real elastic robot?
- How does the learned controller exploit the elastic actuators?
- What is the added value of closing the loop with RL?
- Is the open-loop controller needed or is RL from scratch enough?

Setup

The agent is trained at 30Hz but can be evaluated at 60Hz (limited by the tracking system). The oscillators parameters are optimized during 45 minutes for the trotting experiment and 25 minutes for the pronking one (250 and 160 trials respectively). The RL controller is then trained on top during one hour. For safety, the motor velocities are capped at $\dot{\theta}_{\max} = 4\text{rad/s}$.

Hand-tuned Baseline. We use parameters tuned by hand by a human experimenter familiar with the elastic quadruped as a baseline. The experimenter was given the same time and search space as the BBO algorithm, as well as an explanation of the meaning and importance of each parameter.

RL Baseline. When learning from scratch, we use the setting described in [SKL23] to be able to learn a controller in minutes, using 10 gradient steps per control step. The action space is extended to be similar in task space to the one used by the open-loop controller.

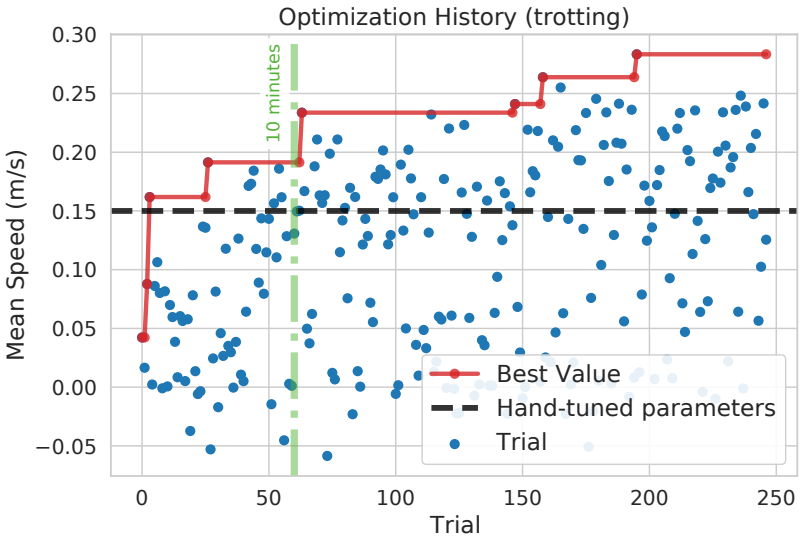


Figure 6.11: Optimization history plot for trotting (45 minutes).

Fast Trotting

Without the need for a simulator or prior knowledge, the automatic tuning with BBO quickly finds good parameters for trotting in less than an hour with the real robot. Looking

at the optimization history in Fig. 6.11, the algorithm discovers parameters that match the hand-tuned performance in less than 5 minutes (30 trials, a trial takes around 10s on average, accounting for resets) and parameters that exceed 0.20m/s in 10 minutes (60 trials).

After only 40 minutes of optimization, the gains are significant for the trotting gait (cf. Table 6.4): with the same maximum motor velocity, the quadruped trots 70% faster, reaching a mean speed of 0.26m/s (vs. 0.15m/s with hand-tuned parameters).

Closing the loop with reinforcement learning helps to further improve performance. Both with the hand-tuned and the optimized oscillators, the robot trots significantly faster (20% and 30% faster respectively). The combination of optimized oscillators + RL gives the best results (0.34m/s), and the hand-tuned oscillators + RL do not reach the performance of the optimized oscillators gait (0.19m/s).

	RL	Oscillators		Oscillators + RL	
		hand-tuned	optimized	hand-tuned	optimized
Speed (m/s) ↑	0.14	0.15	**0.26**	0.19	**0.34**
Mean $\mathcal{R}^{\frac{\dot{q}}{\dot{\theta}}}$ ↑	1.4 +/- 0.1	1.4 +/- 0.1	**1.9 +/- 0.1**	1.6 +/- 0.1	**2.0 +/- 0.3**

Table 6.4: Results for the fast trotting experiment. The optimized gait trots 70% faster that the hand-tuned controller and exploit more the springs.

Concerning joint and motor velocities, the optimized controller reaches higher peak joint velocities for the same peak motor velocities (Table 6.4). The peak joints velocities are almost twice the one of the motors ($\mathcal{R}^{\frac{\dot{q}}{\dot{\theta}}}$=2.0 vs. 1.4 for the hand-tuned) showing the potential of springs for locomotion. In fact, as seen in Fig. 6.12, the peak in joint velocity $\dot{q}$ corresponds to a sudden decrease in the spring potential energy: the energy of the spring is converted into kinetic energy. The main difference between the results of hand-tuned and optimized parameters is the timing of this conversion. For the learned controller, it happens when the motor velocity $\dot{\theta}$ is still at its peak [B], while for the hand-tuned one, it happens when the motor velocity has already started to decrease [A].

Is RL from scratch sufficient?

Table 6.4 also shows the result of learning to walk from scratch with RL on the real robot. The RL controller learns to walk in 10 minutes only, but the performance plateaus afterward. The RL controller barely reaches the hand-tuned oscillators performance (0.14m/s), and this translates into less use of the springs ($\mathcal{R}^{\frac{\dot{q}}{\dot{\theta}}} \approx 1.4$).

This experiment illustrates the shortcomings of learning from scratch. RL would require extensive reward engineering [LHW+20, MLH+22] to achieve natural-looking gaits similar to the open-loop oscillators (e. g. adding a foot clearance reward). The gait learned by RL is also unpredictable: since we only optimize for the forward speed, it does not have to trot, but could for example, pronk. On the other hand, as shown in Section 6.1, the gaits produced by the open-loop controller encode basic primitives for walking (e. g. foot clearance does not need to be specified in the reward), and the phase shifts gives control over the gait type. Learning from scratch with RL is more dangerous for the hardware, as any kind of movement of the legs is allowed, and the smoothness of the controller must be enforced to not damage the motors with high frequency control output (as explored in

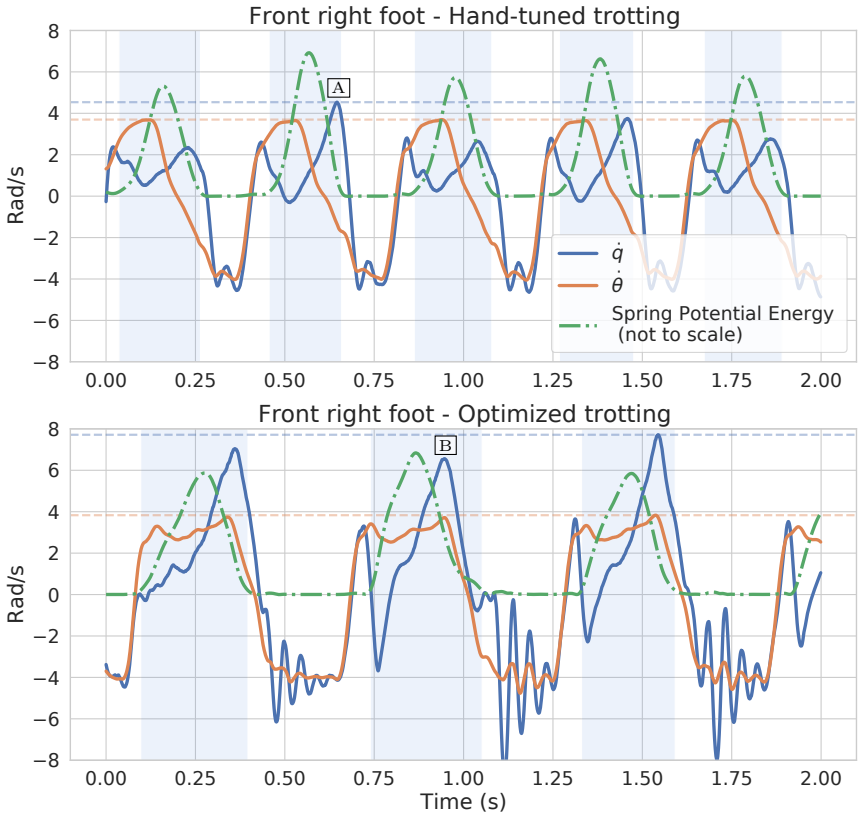


Figure 6.12: Joint and motor velocity of the front right foot while trotting for a 2-second period. Darker background represents stance phase. Blue and orange dotted lines represent maximum joint and motor velocities, respectively.

Chapter 4).

Stabilizing Pronking

For the pronking task, the BBO algorithm discovers multiple good sets of parameters in less than 10 minutes (60 trials), as seen in the optimization history in Fig. 6.13. The hand-tuned and optimized oscillators parameters both allow the robot to jump in place, but the open-loop controller rapidly fails when the robot roll angle increases, either as a consequence of an applied perturbation or due to the interaction with the floor.

By closing the loop between the robot state and actions, the reinforcement learning controller helps mitigating this issue, as summarized in Table 6.5. Here we see that the solution with RL results in no failures. The quadruped also jumps better in place (less drift with respect to the starting position) while keeping the same jumping height.

As for the trotting experiment, adding RL on top of the hand-tuned oscillators improves performance but does not match the optimized controller.

We show in Fig. 6.10 the patterns for the hand-tuned and optimized pronking gaits. The optimized parameters are quite different from the hand-tuned ones: the swing phase is shorter (0.10s vs. 0.18s for the hand-tuned one) and stance phase longer (0.22s vs. 0.15s), and the ratio between the two is inverted (swing phase shorter than the stance phase).

The energy evolution in Fig. 6.14 demonstrates how elastic properties of the actuators are exploited. The robot first crouches on the floor (decrease in gravitational potential

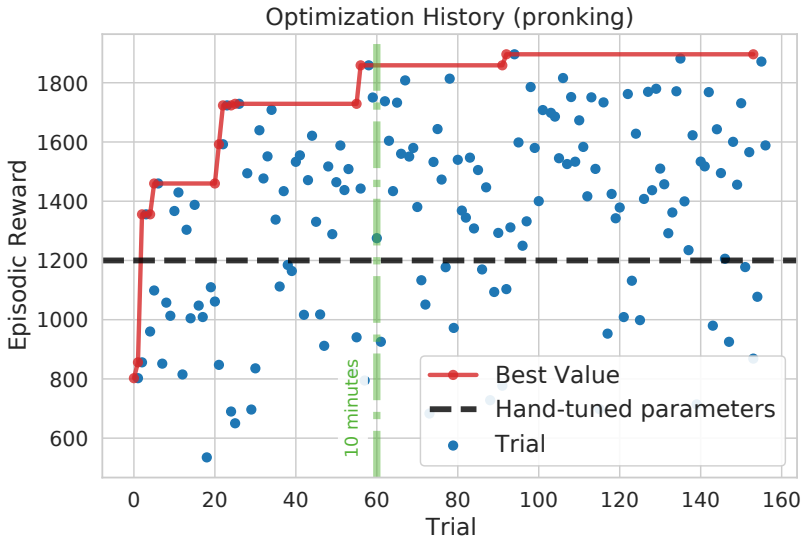


Figure 6.13: Optimization history plot for pronking (30 minutes).

energy), compressing and loading the spring C. To jump, it then pushes all its legs against the floor, while at the same time releasing the energy stored in the springs. This energy is converted both into kinetic energy (the robot reaches its maximum speed shortly after takeoff D) and gravitational potential energy. After reaching the maximum height E, the gravitational potential energy is converted back into kinetic energy before the robot lands and the jumping cycle starts again.

The main difference between the hand-tuned and optimized parameters is the maximum spring potential energy. This results in higher jumps for the optimized controller (higher peak in gravitational potential energy in Fig. 6.14).

Finally, the peak joint velocity is higher for pronking compared to trotting ($\mathcal{R}^{\dot{q}}_{\dot{\theta}}$=3.5 vs. 1.8) because the leg springs are compressed and released synchronously. As shown in Table 6.5, the peak joint velocity is on average 3.5 times the maximum velocity reached by the motors for the optimized parameters, versus 2.9 for the hand-tuned parameters.

	Oscillators		Oscillators + RL	
	hand-tuned	optimized	hand-tuned	optimized
Mean reward (10^3) ↑	1.0 +/- 0.1	1.4 +/- 0.2	1.4 +/- 0.1	**1.6 +/- 0.1** (+60%)
Drift Δ_{xy} cost ↓	1.3 +/- 0.1	1.5 +/- 0.1	1.3 +/- 0.1	1.3 +/- 0.1
Angular vel. cost ↓	204 +/- 6	194 +/- 14	183 +/- 10	**160 +/- 10**
Max height (cm) ↑	7.3	9.3	8.1	9.4
Failures	1 failure	1 failure	no failure	no failure
Mean $\mathcal{R}^{\dot{q}}_{\dot{\theta}}$ ↑	2.9 +/- 0.2	**3.3 +/- 0.2**	3.2 +/- 0.2	**3.5 +/- 0.3** (+20%)

Table 6.5: Results for the jumping in place experiment, on 5 evaluation episodes. Failures occur without external perturbations.

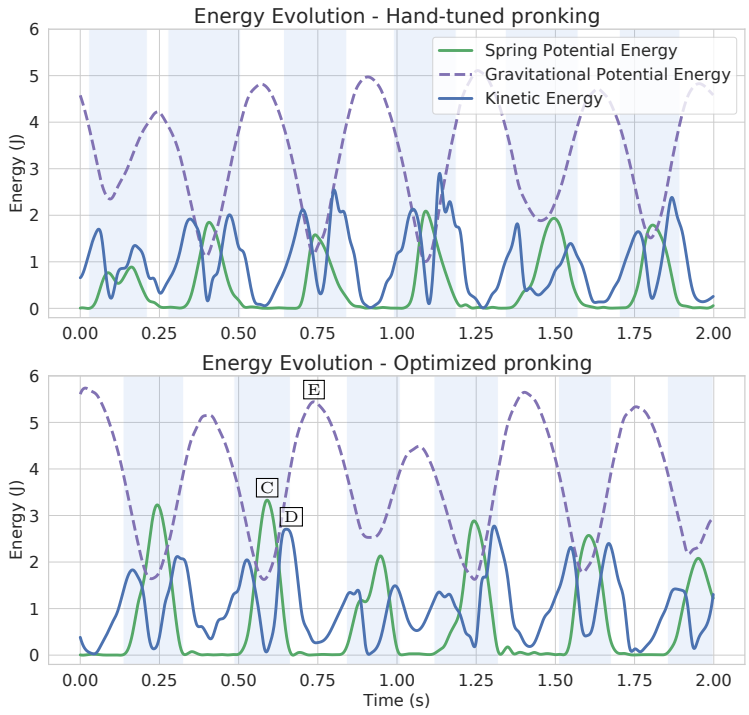


Figure 6.14: Energy evolution while pronking for a 2-second period.

6.3 Learning from Human Feedback

Designing a reward function to achieve a desired behavior can be a tedious and time-consuming process, often requiring many trial and error [DVM+14, SYH+19]. In the case of on-robot learning, obtaining a clean reward signal can be particularly challenging. For instance, in the last section (and in Chapter 4), a tracking system surrounding a treadmill was needed to reward the agent for moving forward. Outside the treadmill, an IMU combined with leg kinematics can be used to estimate the robot's motion, but the signal is noisier and subject to drift [BHH+08].

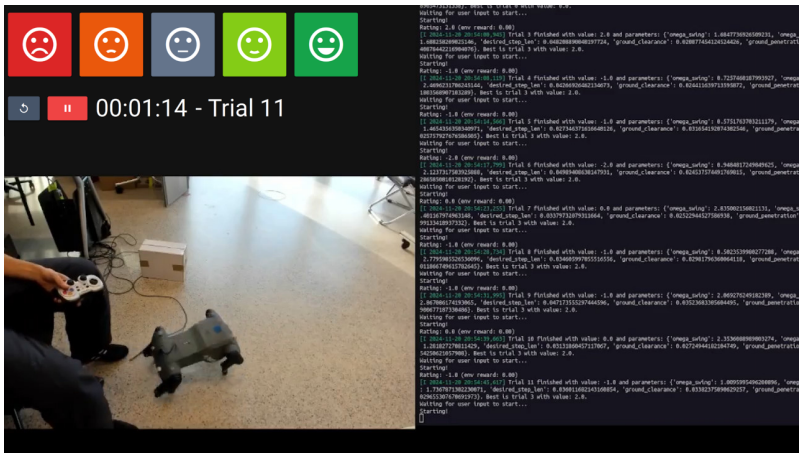


Figure 6.15: Learning from human feedback directly on the real robot with a web-based GUI (top left)

To address this, we investigate replacing engineered reward by a simple human feedback [ISK+01, VH18] to optimize the gaits generated by open-loop oscillators (Section 6.2.1). A human observer evaluates the robot's behavior via a web-based GUI (top left of Fig. 6.15) and provides a discrete rating (five possible values, from -2 to 2) for each trial. This rating serves as the reward signal for the CMA-ES algorithm (with a population size of 10), which optimizes the oscillator parameters. To improve efficiency, a trial is stopped as soon as the human clicks on one of the rating buttons.

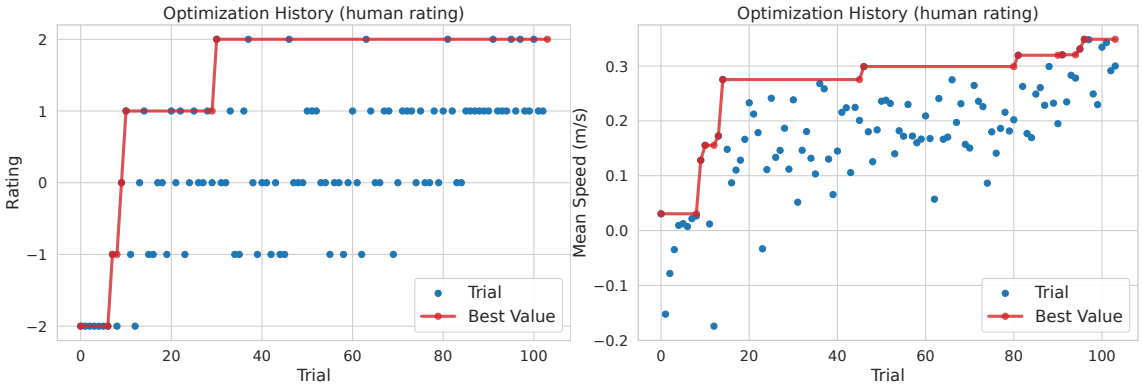


Figure 6.16: Learning curve in simulation using only human feedback (left) and the corresponding forward speed for each trial (right). The forward speed is not available to the robot during training.

In the left panel of Fig. 6.16, we show the learning curve in simulation and display the ground truth speed in the right panel (not available to the robot during training). The robot learns to walk forward in five minutes ($\approx$ 50 trials), using only human feedback. This experiment has been successfully reproduced five times, directly on the real robot, with different human labelers (Fig. 6.15 shows a screenshot of one of the experiment).

6.4 Conclusion

In this chapter, we have explored the benefits of guiding RL with prior knowledge to learn locomotion controllers directly on a real quadruped robot.

We have seen that open-loop oscillators can serve as a useful starting point for learning locomotion controllers, providing insight into the current limitations of DRL algorithms.

By integrating the open-loop oscillators with RL, we can further improve performance and robustness, learning directly on the *bert* quadruped. This approach eliminates the need for complex reward engineering and massively parallel simulation. In particular, the learned gaits exploit the natural dynamics of the robot, enabling efficient energy storage and release in the springs, without requiring explicit reward design.

Furthermore, a learning controller can adapt to new situations, such as changes in dynamics, and discover new behaviors. This ability to quickly adapt and find innovative gaits was demonstrated in an experiment with the ISS.

6.4.1 Epilogue: Surface Avatar Mission with the International Space Station

The Surface Avatar experiment series [LSL+22, SSL+24] is a telerobotics mission in which an astronaut on the International Space Station (ISS) commands robots on Earth with varying degrees of autonomy. For this mission, the elastic quadruped robot *bert* was heavily modified. The new robot, called *norbert* (shown in Fig. 6.17), has new motors, springs with a different stiffness ($k \approx 3.1$Nm/rad vs. 2.75Nm/rad previously), a 400g battery, and a 1kg arm mounted on top of it. These modifications significantly change the center of mass, symmetries, and dynamics of the robot, making it impossible to reuse any of the gaits that were optimized for *bert*.

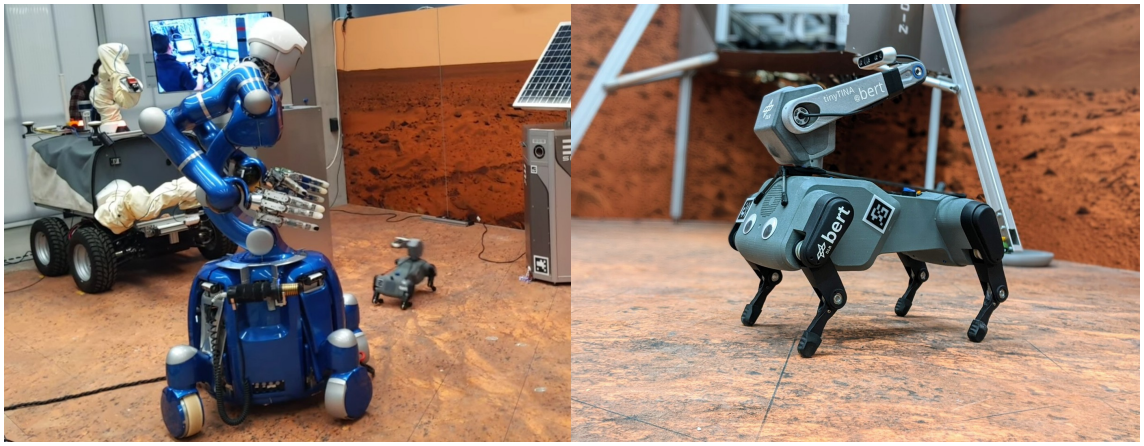


Figure 6.17: Surface Avatar mission with the ISS, the astronaut remotely operating the robot can be seen in the TV (left) and the modified hardware, *norbert* (right)

For the mission, *norbert* needed to walk forward, backward, and turn [SSL+24]. However, unlike other quadruped robots, *norbert* has no hip joints, making it necessary to find a completely new strategy for turning. This is where learning directly with a model-free approach on the real hardware was crucial. We used the open-loop controller in task space described in this chapter and enabled the agent to discover novel behavior by learning the phase shift φ_i between oscillators. With just 30 minutes of retraining for each gait, we were able to provide the desired commands to the astronaut, making *norbert* the first quadruped robot to be controlled from space.

CHAPTER 7

Conclusion

In this thesis, we have focused on developing techniques to enable deep reinforcement learning on real robots. As our conclusion, we summarize the main contributions of this thesis, its impact on the DRL field, discuss the remaining open problems to extend the presented research, and offer a perspective on how RL for real robots might look like in the future.

7.1 Summary

7.1.1 TL;DR

Throughout this thesis, we have applied reinforcement learning directly on real robots. This was made possible by reliable, fast software as well as reproducible experiments (Chapter 3). Learning from scratch can be dangerous and inefficient, so we have explored different methods to incorporate expert knowledge and guide the learning agent.

In Chapter 4, we developed a safer and smoother exploration strategy that allows learning in the real world without any modifications. In addition, we showed how to leverage different types of expert knowledge for two different robots with elastic elements.

For the *David* elastic neck (Chapter 5), we integrated a new pose estimation model to obtain a feedforward controller, and reformulating the problem as a task-conditioned one led to improved results. In the case of the elastic quadruped *bert* (Chapter 6), we combined RL with open-loop oscillators, which were shown to be tailored for locomotion.

In all cases, enabling RL on real robots with expert knowledge reduced training time, wear-and-tear, and improved the final performance.

7.1.2 Challenges of Real-Robot Learning

In the introduction (Section 1.2), we presented several challenges that arise when learning directly on a real robot. We now summarize how the different contributions of this dissertation address those challenges and enable on-robot RL.

Challenge i. (Exploration-Induced Wear and Tear)

gSDE (Chapter 4) is specifically designed to tackle the problem of wear and tear by smoothing the exploration process. The standard approach of adding step-based Gaussian noise to the actions can lead to jerky movements, causing unnecessary stress on the hardware. In contrast, gSDE samples exploration parameters less frequently and uses policy features, resulting in smoother exploration. This directly addresses the challenge of minimizing mechanical fatigue during the learning process.

Integrating a data-driven, fault-tolerant pose estimator for the *David* elastic neck (Chapter 5) reduces the need for extensive exploration. When inverted, the pose estimator provides a feedforward controller that can guide the robot to desired poses, reducing reliance on random exploration and minimizing the risk of wear and tear.

For the *bert* quadruped, using open-loop oscillators as a basis for locomotion control (Chapter 6) inherently generates smooth, rhythmic movements. This approach reduces the reliance on random exploration to discover basic locomotion patterns, further minimizing wear and tear on the hardware.

Challenge ii. (Sample Efficiency)

Stable-Baselines3 (SB3), SBX, and the RL Zoo (Chapter 3) directly address the need for sample efficiency in real-robot learning. By providing reliable and tested implementations of DRL algorithms, they ensure that experiments are reproducible and minimize wasted trials due to bugs or implementation errors. Additionally, SBX, with its focus on speed, allows for more gradient updates per unit of time, improving sample efficiency.

Two methods are presented to improve sample efficiency in controlling the *David* elastic neck (Chapter 5). Goal-conditioned RL with Hindsight Experience Replay (HER) relabels unsuccessful experiences with alternative goals, allowing the agent to learn from a wider range of situations. Combining RL with a feedforward controller derived from the pose estimator enables the agent to start with a reasonably good policy, drastically reducing the number of trials needed to achieve high performance.

For the *bert* quadruped (Chapter 6), the open-loop oscillators provide a strong prior for locomotion control, reducing the search space for the RL agent. This allows *bert* to learn effective locomotion gaits with significantly fewer interactions than learning from scratch (Chapter 4).

Challenge iii. (Real-Time Constraints)

Both Stable-Baselines3 and SBX (Chapter 3) are designed to satisfy the real-time needs of robot control. SB3 and its RL Zoo provide a stable and well-documented framework that can be used to run experiments over several days, while SBX focuses on speed, using the Jax library for faster execution. The just-in-time compilation of Jax allow policy updates to be performed in a timely manner, meeting the real-time constraints of a robot control loop.

Challenge iv. (Computational Resource Constraints)

With a focus on clean, efficient code, SB3 and SBX models (Chapter 3) can be deployed on the resource-constrained hardware often used in robotics (with the help of ONNX in

the case of SB3 [1]). This makes it possible to run DRL algorithms directly on the robot, without requiring powerful external computers.

The data-driven pose estimation method for the *David* elastic neck (Chapter 5) uses simple models (linear and polynomial) that are computationally efficient. This ensures that pose estimation can be performed in real-time on the robot's hardware. This efficiency is maintained even when managing an ensemble of estimators.

Open-loop oscillators are computationally lightweight, requiring minimal resources to generate control signals. This allows for real-time control even on embedded platforms with limited processing power. The on-robot learning experiments with the *norbert* quadruped, a modified version of *bert* used in a mission with the International Space Station, highlights this capability.

7.1.3 Impact

The work presented in this thesis has already impacted the field of deep reinforcement learning, particularly in its application to real-world robotics.

High Quality Software The tools developed during this thesis and presented in Chapter 3 (SB3, SBX, and the RL Zoo) have become a standard resource. They are used by researchers and practitioners that want to apply RL in various domains, from climate change [LCNB22] and particle accelerators [VGK+23] to video games [Whi23] and drone racing [SRM+23]. The impact of SB3 extends beyond research, as it is also integrated into educational resources [SS23, RFK+23, Car22], benefiting both students and educators.

Enabling RL for Robots The methods presented have seen broader application beyond the scope of this dissertation. Generalized State-Dependent Exploration (gSDE) has been applied to other robotic platforms [SHI+23, dPLF23] and inspired further research [EHPM22, CMVHM23], contributing to a winning robotic manipulation solution [CMVHM23]. The pose estimation and control approach for the *David* elastic neck (Chapter 5) was applied to a new neck of the neo*David* platform [WHB+23] and integrated with vision for robust arm tracking [Sto23]. Finally, the open-loop oscillators for locomotion were used in two telerobotics missions with the International Space Station [LSL+22, SSL+24].

7.2 Future Work

Software SB3, SBX, and the RL Zoo are powerful tools for conducting single-agent, model-free RL experiments. However, they do not cover multi-agent, model-based, or offline RL, all of which are missing easy-to-use and reliable libraries. In addition, the RL Zoo currently lacks features to analyze and group experiments independently of an online service, and would benefit from a GUI. Finally, deploying a trained model with SB3 (using ONNX or C++) currently requires manual steps, and automating this part would benefit the community.

Safety In this thesis, we have made efforts to prioritize safety by using safer exploration techniques and incorporating expert knowledge into the task design. However, it is still

[1] `https://github.com/onnx/onnx`

possible for the agent to encounter dangerous situations, and in some experiments, a human was required to intervene and prevent harm (e. g. when learning from scratch, *bert* tended to load its springs and attempt a backflip). Instead of relying on soft constraints or relying on a human, one line of research is to explore strategies to ensure that the agent respects hard constraints and never explores dangerous regions of the space.

Pose Estimation and Iterative Learning. The fault-tolerant pose estimation method presented in this thesis is able to handle many failures, but could benefit from a more sophisticated ensembling technique to improve its scalability. Additionally, the method requires an initial dataset that covers the entire task space in order to ensure accurate pose predictions in all situations. A possible solution to these limitations is to use an iterative learning approach for both the pose estimation and the controller. Initially, the controller would only be able to reach a small region around the default pose, where the initial pose estimator would be accurate. Over time, as the pose prediction and controller accuracy improve, this region would expand until it covers the entire domain.

Open-Loop Oscillators. While our approach generates desired joint positions using oscillators without relying on the robot state, a PD controller is still required in some environments to convert these positions into torque commands. Since the generated torques appear to be periodic, a straightforward extension would be to replace the PD by additional oscillators (additional harmonic terms).

Learning locomotion. In this thesis, we have shown that it is possible to quickly learn a locomotion controller for an elastic quadruped robot. However, the current approach is limited to single-task controllers and has primarily been tested on flat ground. Future work should focus on learning command-conditioned controllers and adapting the gait to new and unseen environments, including outdoor environments. This would allow the robot to perform a wider range of tasks and operate in a variety of real-world settings.

7.3 Outlook

This thesis is a step towards bringing deep reinforcement learning to real robots. Although many challenges remain, learning methods have started to be integrated into robotic controllers, such as in the Anymal robot and more recently in Spot, which historically focused on model-based MPC control [HGJ+16, Dyn20].

Learning from Human Feedback

One important aspect of learning controllers is reward design. Designing reward functions for complex robotic tasks can be challenging, as it requires carefully specifying the desired behavior and balancing multiple objectives. Misspecified rewards can lead to unintended (e. g. reward hacking, the agent maximizes the objective but does not solve the task) and potentially unsafe behaviors.

The recent interest in learning from human feedback [KS08, CLB+17], as seen in the training of large language models [SOW+20, OWJ+22], should benefit RL for robotics. By learning reward functions from human preferences, the robot could learn the desired behavior without the need for manual reward engineering. However, integrating human

feedback into RL for robots remains an open research question, requiring further investigation into effective feedback mechanisms, human-robot interaction, and robustness to feedback noise and bias.

Advances in Software, Hardware and Cost of Generality

RL practitioners, and more generally machine learning practitioners, often overlook advancements in hardware design. Recent successes in learning controllers on real robots are not only due to better simulators, algorithms, or compute power but also to more robust hardware. As RL learns by trial and error, it is demanding on the robot.

To improve performance, one can either enhance software or hardware, or both (co-design). The trade-off between generality and specificity is also crucial, as shown in the Amazon Picking Challenge, where the compromise between embodiment and computation was a key aspect [EHJ+16]. In this challenge, it was more robust and easier to use suction to pick objects instead of a more versatile robotic hand. Similarly, in the case of folding clothes, one can use a dual-arm setup with complex interactions or create a system that solves the problem by design, such as a clothes-folding board.

The Role of Simulation and Learning on Real Robots

As simulation can now be easily scaled to thousands of robots on a single GPU [RHRH22], it will likely remain the starting point when designing learning controllers, as long as the robot can be modeled accurately enough. However, learning in simulation should be seen as an initial step to obtain a pre-trained controller. Just as pre-trained encoders are used for vision tasks, learning in simulation should enable pre-trained controllers that can be quickly fine-tuned for new tasks or robots.

To further adapt to new situations, compensate for robot defects, or improve accuracy, simulation alone is not enough. Fine-tuning in the real world to account for model inaccuracies or to adapt to new situations (e. g. robot damage, unseen terrain) will hopefully become more common. To improve the safety and tackle common issues of RL, such as credit assignment, it should be combined with classic robotics techniques like motion planners for higher-level control, obstacle avoidance, and long-term path planning.

Ultimately, the future of robot learning lies in combining the flexibility of simulation with the directness of on-robot training, enabling robots to learn quickly, adapt to new situations, and perform complex tasks reliably.

Bibliography

[AAC+19] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[ABC+20] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[Ach18] Joshua Achiam. Spinning up in deep reinforcement learning. `https://github.com/openai/spinningup`, 2018.

[AM08] Karl Johan Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA, 2008.

[APSI13] Mostafa Ajallooeian, Soha Pouya, Alexander Sproewitz, and Auke J Ijspeert. Central pattern generators augmented with virtual model control for quadruped rough terrain locomotion. In *2013 IEEE international conference on robotics and automation*, pages 3321–3328. IEEE, 2013.

[ASC+21] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

[ASDS20] Alin Albu-Schäffer and Cosimo Della Santina. A review on nonlinear modes in conservative mechanical systems. *Annual Reviews in Control*, 50:49–71, 2020.

[ASY+19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2623–2631, New York, NY, USA, 2019. Association for Computing Machinery.

[AWR+17] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Neural Information Processing Systems*, pages 5048–5058, 2017.

[BDM17] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[BDWN07] David Braganza, Darren M Dawson, Ian D Walker, and Nitendra Nath. A neural network controller for continuum robots. *IEEE Trans. on Robotics*, 23(6):1270–1277, 2007.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[BGC+22] Dieter Büchler, Simon Guist, Roberto Calandra, Vincent Berenz, Bernhard Schölkopf, and Jan Peters. Learning to play table tennis from scratch using muscular robots. *IEEE Transactions on Robotics*, 2022.

[BHH+08] Michael Bloesch, Marco Hutter, Mark A. Hoepflinger, Stefan Leutenegger, Christian Gehring, C. David Remy, and Roland Siegwart. State estimation for legged robots - consistent fusion of leg kinematics and imu. In *Robotics: Science and systems*, 2008.

[BI22] G. Bellegarda and A. J. Ijspeert. CPG-RL: Learning central pattern generators for quadruped locomotion. *IEEE Robotics and Automation Letters*, 2022.

[BLA+23] Filip Bjelonic, Joonho Lee, Philip Arm, Dhionis Sako, Davide Tateo, Jan Peters, and Marco Hutter. Learning-based design and control for quadrupedal robots with parallel-elastic actuators. *IEEE Robotics and Automation Letters*, 2023.

[BPB+24] Aditya Bhatt, Daniel Palenicek, Boris Belousov, Max Argus, Artemij Amiranashvili, Thomas Brox, and Jan Peters. Cross$q$: Batch normalization in deep reinforcement learning for greater sample efficiency and simplicity. In *The Twelfth International Conference on Learning Representations*, 2024.

[Bre96] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[BS01] J Andrew Bagnell and Jeff G Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 1615–1620. IEEE, 2001.

[BSA83] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[BSO+16] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in neural information processing systems*, pages 1471–1479, 2016.

[Car22] Stéphane Caron. Open-source wheeled biped robots. `https://github.com/upkie/upkie`, 2022.

[CB21] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

[CCS09] D.B. Camarillo, C.R. Carlson, and J.K. Salisbury. Configuration tracking for continuum manipulators with coupled tendon drive. *IEEE Trans. on Robotics*, 25(4):798–808, 2009.

[CI08] Alessandro Crespi and Auke Jan Ijspeert. Online optimization of swimming and crawling in an amphibious snake robot. *IEEE Transactions on Robotics*, 24(1):75–87, 2008.

[CLB+17] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[CLNE17] Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. Reinforcement learning Coach. `https://doi.org/10.5281/zenodo.1134899`, December 2017.

[CMG+18] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A research framework for deep reinforcement learning. arXiv: 1812.06110v1 [cs.LG], 2018.

[CMVHM23] Alberto Silvio Chiappa, Alessandro Marin Vargas, Ann Huang, and Alexander Mathis. Latent exploration for reinforcement learning. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 56508–56530. Curran Associates, Inc., 2023.

[COR+24] Open X-Embodiment Collaboration, Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, Albert Tung, Alex Bewley, Alex Herzog, Alex Irpan, Alexander Khazatsky, Anant Rai, Anchit Gupta, Andrew Wang, Andrey Kolobov, Anikait Singh, Animesh Garg, Aniruddha Kembhavi, Annie Xie, Anthony Brohan, Antonin Raffin, Archit Sharma, Arefeh Yavary, Arhan Jain, Ashwin Balakrishna, Ayzaan Wahid, Ben Burgess-Limerick, Beomjoon Kim, Bernhard Schölkopf, Blake Wulfe, Brian Ichter, Cewu Lu, Charles Xu, Charlotte Le, Chelsea Finn, Chen Wang, Chenfeng Xu, Cheng Chi, Chenguang Huang, Christine Chan, Christopher Agia, Chuer Pan, Chuyuan Fu, Coline Devin, Danfei Xu,

Daniel Morton, Danny Driess, Daphne Chen, Deepak Pathak, Dhruv Shah, Dieter Büchler, Dinesh Jayaraman, Dmitry Kalashnikov, Dorsa Sadigh, Edward Johns, Ethan Foster, Fangchen Liu, Federico Ceola, Fei Xia, Feiyu Zhao, Felipe Vieira Frujeri, Freek Stulp, Gaoyue Zhou, Gaurav S. Sukhatme, Gautam Salhotra, Ge Yan, Gilbert Feng, Giulio Schiavi, Glen Berseth, Gregory Kahn, Guangwen Yang, Guanzhi Wang, Hao Su, Hao-Shu Fang, Haochen Shi, Henghui Bao, Heni Ben Amor, Henrik I Christensen, Hiroki Furuta, Homer Walke, Hongjie Fang, Huy Ha, Igor Mordatch, Ilija Radosavovic, Isabel Leal, Jacky Liang, Jad Abou-Chakra, Jaehyung Kim, Jaimyn Drake, Jan Peters, Jan Schneider, Jasmine Hsu, Jeannette Bohg, Jeffrey Bingham, Jeffrey Wu, Jensen Gao, Jiaheng Hu, Jiajun Wu, Jialin Wu, Jiankai Sun, Jianlan Luo, Jiayuan Gu, Jie Tan, Jihoon Oh, Jimmy Wu, Jingpei Lu, Jingyun Yang, Jitendra Malik, Jo ao Silvério, Joey Hejna, Jonathan Booher, Jonathan Tompson, Jonathan Yang, Jordi Salvador, Joseph J. Lim, Junhyek Han, Kaiyuan Wang, Kanishka Rao, Karl Pertsch, Karol Hausman, Keegan Go, Keerthana Gopalakrishnan, Ken Goldberg, Kendra Byrne, Kenneth Oslund, Kento Kawaharazuka, Kevin Black, Kevin Lin, Kevin Zhang, Kiana Ehsani, Kiran Lekkala, Kirsty Ellis, Krishan Rana, Krishnan Srinivasan, Kuan Fang, Kunal Pratap Singh, Kuo-Hao Zeng, Kyle Hatch, Kyle Hsu, Laurent Itti, Lawrence Yunliang Chen, Lerrel Pinto, Li Fei-Fei, Liam Tan, Linxi "Jim" Fan, Lionel Ott, Lisa Lee, Luca Weihs, Magnum Chen, Marion Lepert, Marius Memmel, Masayoshi Tomizuka, Masha Itkina, Mateo Guaman Castro, Max Spero, Maximilian Du, Michael Ahn, Michael C. Yip, Mingtong Zhang, Mingyu Ding, Minho Heo, Mohan Kumar Srirama, Mohit Sharma, Moo Jin Kim, Naoaki Kanazawa, Nicklas Hansen, Nicolas Heess, Nikhil J Joshi, Niko Suenderhauf, Ning Liu, Norman Di Palo, Nur Muhammad Mahi Shafiullah, Oier Mees, Oliver Kroemer, Osbert Bastani, Pannag R Sanketi, Patrick "Tree" Miller, Patrick Yin, Paul Wohlhart, Peng Xu, Peter David Fagan, Peter Mitrano, Pierre Sermanet, Pieter Abbeel, Priya Sundaresan, Qiuyu Chen, Quan Vuong, Rafael Rafailov, Ran Tian, Ria Doshi, Roberto Mart'in-Mart'in, Rohan Baijal, Rosario Scalise, Rose Hendrix, Roy Lin, Runjia Qian, Ruohan Zhang, Russell Mendonca, Rutav Shah, Ryan Hoque, Ryan Julian, Samuel Bustamante, Sean Kirmani, Sergey Levine, Shan Lin, Sherry Moore, Shikhar Bahl, Shivin Dass, Shubham Sonawani, Shuran Song, Sichun Xu, Siddhant Haldar, Siddharth Karamcheti, Simeon Adebola, Simon Guist, Soroush Nasiriany, Stefan Schaal, Stefan Welker, Stephen Tian, Subramanian Ramamoorthy, Sudeep Dasari, Suneel Belkhale, Sungjae Park, Suraj Nair, Suvir Mirchandani, Takayuki Osa, Tanmay Gupta, Tatsuya Harada, Tatsuya Matsushima, Ted Xiao, Thomas Kollar, Tianhe Yu, Tianli Ding, Todor Davchev, Tony Z. Zhao, Travis Armstrong, Trevor Darrell, Trinity Chung, Vidhi Jain, Vincent Vanhoucke, Wei Zhan, Wenxuan Zhou, Wolfram Burgard, Xi Chen, Xiangyu Chen, Xiaolong Wang, Xinghao Zhu, Xinyang Geng, Xiyuan Liu, Xu Liangwei, Xuanlin Li, Yansong Pang, Yao Lu, Yecheng Jason Ma, Yejin Kim, Yevgen Chebotar, Yifan Zhou, Yifeng Zhu, Yilin Wu, Ying Xu, Yixuan Wang, Yonatan Bisk, Yongqiang Dou, Yoonyoung Cho, Youngwoon Lee, Yuchen Cui, Yue Cao, Yueh-Hua Wu, Yujin Tang, Yuke Zhu, Yunchu Zhang, Yunfan Jiang,

Yunshuang Li, Yunzhu Li, Yusuke Iwasawa, Yutaka Matsuo, Zehan Ma, Zhuo Xu, Zichen Jeff Cui, Zichen Zhang, Zipeng Fu, and Zipeng Lin. Open X-Embodiment: Robotic learning datasets and RT-X models. In *41th IEEE International Conference on Robotics and Automation, ICRA 2024*. IEEE, 2024.

[CSO18] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. Gep-pg: Decoupling exploration and exploitation in deep reinforcement learning algorithms. In *International conference on machine learning*, pages 1039–1048. PMLR, 2018.

[CSPD16] Roberto Calandra, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. Bayesian optimization for learning gaits under uncertainty. *Annals of Mathematics and Artificial Intelligence*, 76(1):5–23, 2016.

[CVS+19] Konstantinos Chatzilygeroudis, Vassilis Vassiliades, Freek Stulp, Sylvain Calinon, and Jean-Baptiste Mouret. A survey on policy search algorithms for learning robot controllers in a handful of trials. *IEEE Transactions on Robotics*, 36(2):328–347, 2019.

[CW80] Avis H Cohen and Peter Wallén. The neuronal correlate of locomotion in fish: "fictive swimming" induced in an in vitro preparation of the lamprey spinal cord. *Experimental brain research*, 41(1):11–18, 1980.

[CWZR21] Xinyue Chen, Che Wang, Zijian Zhou, and Keith W. Ross. Randomized ensembled double q-learning: Learning fast without a model. In *International Conference on Learning Representations*, 2021.

[DALM+20] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. An empirical investigation of the challenges of real-world reinforcement learning. *arXiv preprint arXiv:2003.11881*, 2020.

[DALM+21] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.

[DCH+16] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[DCR+19] Bastian Deutschmann, Maxime Chalon, Jens Reinecke, Maximilian Maier, and Christian Ott. Six-dof pose estimation for a tendon-driven continuum mechanism without a deformation model. *IEEE Robotics and Automation Letters*, 4(4):3425–3432, 2019.

[DDO17] Bastian Deutschmann, Alexander Dietrich, and Christian Ott. Position control of an underactuated continuum mechanism using a reduced nonlinear model. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 5223–5230. IEEE, 2017.

[Del80] Fred Delcomyn. Neural basis of rhythmic behavior in animals. *Science*, 210(4469):492–498, 1980.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[DNP+13] Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

[dPLF23] Armand du Parc Locmaria and Pierre Fabre. Furuta pendulum: Building and training a rotary inverted pendulum robot. `https://github.com/Armandpl/furuta`, 2023.

[DRBM18] Will Dabney, Mark Rowland, Marc Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[DSBG+17] Cosimo Della Santina, Matteo Bianchi, Giorgio Grioli, Franco Angelini, Manuel Catalano, Manolo Garabini, and Antonio Bicchi. Controlling soft robots: balancing feedback and feedforward elements. *IEEE Robotics & Automation Magazine*, 24(3):75–83, 2017.

[DSLBAS20] Cosimo Della Santina, Dominic Lakatos, Antonio Bicchi, and Alin Albu-Schaeffer. Using nonlinear normal modes for execution of efficient cyclic motions in articulated soft robots. In *International Symposium on Experimental Robotics*, pages 566–575. Springer, 2020.

[DVM+14] Christian Daniel, Malte Viering, Jan Metz, Oliver Kroemer, and Jan Peters. Active reward learning. In *Robotics: Science and systems*, volume 98, 2014.

[Dyn20] Boston Dynamics. Spot robot. `https://www.bostondynamics.com/products/spot`, 2020. Accessed: 2024-04-26.

[Efr92] Bradley Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer, 1992.

[EGW05] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6, 2005.

[EHJ+16] Clemens Eppner, Sebastian Höfer, Rico Jonschkowski, Roberto Martín-Martín, Arne Sieverling, Vincent Wall, and Oliver Brock. Lessons from the amazon picking challenge: Four aspects of building robotic systems. In *Robotics: science and systems*, volume 12, 2016.

[EHPM22] Onno Eberhard, Jakob Hollenstein, Cristina Pinneri, and Georg Martius. Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2022.

[EIS+20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020.

[FAP+18] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. In *International Conference on Learning Representations*, 2018.

[FKMP21] Zipeng Fu, Ashish Kumar, Jitendra Malik, and Deepak Pathak. Minimizing energy consumption leads to the emergence of gaits in legged robots. *Conference on Robot Learning (CoRL)*, 2021.

[FLO+22] Maël Franceschetti, Coline Lacoux, Ryan Ohouens, Antonin Raffin, and Olivier Sigaud. Making reinforcement learning work on swimmer. *arXiv preprint arXiv:2208.07587*, 2022.

[FNKI21] Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, volume 80, pages 1587–1596, 2018.

[Gal77] J. Gall. *Systemantics: How Systems Work and Especially how They Fail.* General systemantics. Quadrangle/New York Times Book Company, 1977.

[Gal16] Yarin Gal. Uncertainty in deep learning. *University of Cambridge*, 1:3, 2016.

[gc19] The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`, 2019.

[GCL+18] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Zhengxing Chen, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. Horizon: Facebook's open source applied reinforcement learning platform. arXiv:1811.00260v5 [cs.LG], 2018.

[GDRS00] P. Goel, G. Dedeoglu, S. I. Roumeliotis, and G. S. Sukhatme. Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 3, pages 2302–2309 vol.3, 2000.

[GDW+20] Adam Gleave, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. Adversarial policies: Attacking deep reinforcement learning. In *International Conference on Learning Representations*, 2020.

[GFB94] Vijaykumar Gullapalli, Judy A Franklin, and Hamid Benbrahim. Acquiring robot skills via reinforcement learning. *IEEE Control Systems Magazine*, 14(1):13–24, 1994.

[GGS00] Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control System Design*. Prentice Hall PTR, USA, 1st edition, 2000.

[GKR+18] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018.

[GLSL16] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*, pages 2829–2838. PMLR, 2016.

[GRC+15] Michele Giorelli, Federico Renda, Marcello Calisti, Andrea Arienti, Gabriele Ferri, and Cecilia Laschi. Neural Network and Jacobian Method for Solving the Inverse Statics of a Cable-Driven Soft Arm With Nonconstant Curvature. *IEEE Trans. on Robotics*, 31(4):823–834, 2015.

[Han09] Nikolaus Hansen. Benchmarking a bi-population cma-es on the bbob-2009 function testbed. In *Proceedings of the 11th annual conference companion on genetic and evolutionary computation conference: late breaking papers*, pages 2389–2396, 2009.

[HDR+22] Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022.

[HDY+22] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and Jo ao G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[HGB+12] Marco Hutter, Christian Gehring, Michael Bloesch, Mark A Hoepflinger, C David Remy, and Roland Siegwart. StarlETH: A compliant quadrupedal robot for fast, efficient, and versatile locomotion. In *Adaptive Mobile Robotics*, pages 483–490. World Scientific, 2012.

[HGF+24] Shengyi Huang, Quentin Gallouédec, Florian Felten, Antonin Raffin, Rousslan Fernand Julien Dossa, Yanxiao Zhao, Ryan Sullivan, Viktor Makoviychuk, Denys Makoviichuk, Mohamad H Danesh, et al. Open rl benchmark: Comprehensive tracked experiments for reinforcement learning. *arXiv preprint arXiv:2402.03046*, 2024.

[HGJ+16] Marco Hutter, Christian Gehring, Dominic Jud, Andreas Lauber, C. Dario Bellicoso, Vassilios Tsounis, Jemin Hwangbo, Karen Bodie, Peter Fankhauser, Michael Bloesch, Remo Diethelm, Samuel Bachmann, Amir Melzer, and Mark Hoepflinger. ANYmal - a highly mobile and dynamic

quadrupedal robot. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 38–44, 2016.

[HHZ+18] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*, 2018.

[HIB+18a] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[HIB+18b] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *The AAAI Conference on Artificial Intelligence*, pages 3207–3214, 2018.

[HIH+22] Takuya Hiraoka, Takahisa Imagawa, Taisei Hashimoto, Takashi Onishi, and Yoshimasa Tsuruoka. Dropout q-functions for doubly efficient reinforcement learning. In *International Conference on Learning Representations*, 2022.

[HKB15] A. Haidu, D. Kohlsdorf, and M. Beetz. Learning action failure models from interactive physics-based simulations. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5370–5375, 2015.

[HKR+22] Shengyi Huang, Anssi Kanervisto, Antonin Raffin, Weixun Wang, Santiago Ontañón, and Rousslan Fernand Julien Dossa. A2c is a special case of ppo. *arXiv preprint arXiv:2205.09123*, 2022.

[HLD+19] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[HLTB01] K. Hamilton, D. Lane, N. Taylor, and K. Brown. Fault diagnosis on autonomous robotic vehicles with recovery: an integrated heterogeneous-knowledge approach. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 4, pages 3232–3237 vol.4, 2001.

[HMK03] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane,

Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[HMVH+18] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[HRE+18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable Baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[HSA+20] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. arXiv: 2006.00979v1 [cs.LG], 2020.

[HTAL17] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning*, volume 70, pages 1352–1361, 2017.

[HUK+14] R. Hornung, H. Urbanek, J. Klodmann, C. Osendorfer, and P. van der Smagt. Model-free robot anomaly detection. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3676–3683, 2014.

[HW07] Q Peter He and Jin Wang. Fault detection using the k-nearest neighbor rule for semiconductor manufacturing processes. *IEEE transactions on semiconductor manufacturing*, 20(4):345–354, 2007.

[HXT+20] Sehoon Ha, Peng Xu, Zhenyu Tan, Sergey Levine, and Jie Tan. Learning to walk in the real world with minimal human effort. *arXiv preprint arXiv:2002.08550*, 02 2020.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870, July 2018.

[HZH+18] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[IAST13] Atil Iscen, Adrian Agogino, Vytas SunSpiral, and Kagan Tumer. Controlling tensegrity robots through evolution. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 1293–1300, New York, NY, USA, 2013. Association for Computing Machinery.

[ICT+18] Atil Iscen, Ken Caluwaerts, Jie Tan, Tingnan Zhang, Erwin Coumans, Vikas Sindhwani, and Vincent Vanhoucke. Policies modulating trajectory generators. In *Conference on Robot Learning*, pages 916–926. PMLR, 2018.

[Ijs08] Auke Jan Ijspeert. Central pattern generators for locomotion control in animals and robots: A review. *Neural Networks*, 21(4):642–653, 2008. Robotics and Neuroscience.

[ISK+01] Charles Isbell, Christian R Shelton, Michael Kearns, Satinder Singh, and Peter Stone. A social reinforcement learning agent. In *Proceedings of the fifth international conference on Autonomous agents*, pages 377–384, 2001.

[JW06] Bryan A Jones and Ian D Walker. Practical kinematics for real-time implementation of continuum robots. *IEEE Trans. on Robotics*, 22(6):1087–1099, 2006.

[KBP13] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[KFPM21] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *Robotics: Science and Systems*, 2021.

[KG17a] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *Advances in neural information processing systems*, pages 5574–5584, 2017.

[KG17b] Wah Loon Keng and Laura Graesser. SLM lab. `https://github.com/kengz/SLM-Lab`, 2017.

[KHJ+19] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8248–8254. IEEE, 2019.

[KIP+18] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on robot learning*, pages 651–673. PMLR, 2018.

[KK18] Eliahu Khalastchi and Meir Kalech. On fault detection and diagnosis in robotic systems. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[KLOAS18] Manuel Keppler, Dominic Lakatos, Christian Ott, and Alin Albu-Schäffer. Elastic structure preserving (esp) control for compliantly actuated robots. *IEEE Transactions on Robotics*, 34(2):317–335, 2018.

[KMVB19] Dmytro Korenkevych, A Rupam Mahmood, Gautham Vasan, and James Bergstra. Autoregressive policies for continuous control deep reinforcement learning. *arXiv preprint arXiv:1903.11524*, 2019.

[Kol18] Sergey Kolesnikov. Accelerated rl. `https://github.com/catalyst-team/catalyst-rl`, 2018.

[KP09] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[KPKP14] Jens Kober, Jan Peters, Jens Kober, and Jan Peters. Movement templates for learning of hitting and batting. *Learning Motor Skills: From Algorithms to Robot Experiments*, pages 69–82, 2014.

[KS04] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 3, pages 2619–2624. IEEE, 2004.

[KS08] W Bradley Knox and Peter Stone. Tamer: Training an agent manually via evaluative reinforcement. In *2008 7th IEEE international conference on development and learning*, pages 292–297. IEEE, 2008.

[KSF17] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a TensorFlow library for applied reinforcement learning. `https://github.com/tensorforce/tensorforce`, 2017.

[KSGV20] Arsenii Kuznetsov, Pavel Shvechikov, Alexander Grishin, and Dmitry Vetrov. Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[LCNB22] Marcus Lapeyrolerie, Melissa S Chapman, Kari EA Norman, and Carl Boettiger. Deep reinforcement learning for conservation decisions. *Methods in Ecology and Evolution*, 13(11):2649–2662, 2022.

[LDRGF18] Timothée Lesort, Natalia Díaz-Rodríguez, Jean-Franois Goudou, and David Filliat. State representation learning for control: An overview. *Neural Networks*, 108:379–392, 2018.

[LFDA16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[LHP+16] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*, 2016.

[LHW+20] Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science robotics*, 5(47):eabc5986, 2020.

[Lin92] Long-Ji Lin. *Reinforcement learning for robots using neural networks.* Carnegie Mellon University, 1992.

[LLN+18] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, 2018.

[LSL+22] Neal Y. Lii, Peter Schmaus, Daniel Leidner, Thomas Krüger, Jessica Grenouilleau, Aaron Pereira, Angelo Giuliano, Adrian S. Bauer, Anne Köpken, Florian Lay, Marco Sewtz, Nicolai Bechtel, Nesrine Batti, Peter Lehner, Samuel Bustamante Gomez, Maximilian Denninger, Werner Friedl, Jörg Butterfass, Edmundo Ferreira, Andrei Gherghescu, Thibaud Chupin, Emiel den Exter, Levin Gerdes, Michael Panzirsch, Harsimran Singh, Ribin Balachandran, Thomas Hulin, Thomas Gumpert, Annika Schmidt, Daniel Seidel, Milan Hermann, Maximilian Maier, Robert Burger, Florian Schmidt, Bernhard Weber, Ralph Bayer, Benedikt Pleintinger, Roman Holderried, Pedro Pavelski, Armin Wedler, Stefan von Dombrowski, Hansjörg Maurer, Martin Görner, Thilo Wüsthoff, Serena Bertone, Thomas Müller, Gerd Söllner, Christian Ehrhardt, Lucia Brunetti, Linda Holl, Mairead Bévan, Robert Mühlbauer, Gianfranco Visentin, and Alin Albu-Schäffer. Introduction to surface avatar: the first heterogeneous robotic team to be commanded with scalable autonomy from the iss. *Proceedings of the International Astronautical Congress, IAC*, 2022-September, 2022.

[LTAP22] Puze Liu, Davide Tateo, Haitham Bou Ammar, and Jan Peters. Robot reinforcement learning on the constraint manifold. In *Conference on Robot Learning*, pages 1357–1366. PMLR, 2022.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[MBT+18] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[MC92] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial intelligence*, 55(2-3):311–365, 1992.

[Met16] k610 CMM Metris. http://www.metris3d.hu, 2016. Accessed: 2020.08.31.

[MGR18] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[MKH21] Nico Mansfeld, Manuel Keppler, and Sami Haddadin. Speed gain in elastic joint robots: An energy conversion-based approach. *IEEE Robotics and Automation Letters*, 6(3):4600–4607, 2021.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. In *Deep Learning Workshop at Neural Information Processing Systems*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[MLH+22] Takahiro Miki, Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics*, 7(62):eabk2822, 2022.

[MMMS21] Siddharth Mysore, Bassel Mabsout, Renato Mancuso, and Kate Saenko. Regularizing action policies for smooth control with reinforcement learning. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1810–1816. IEEE, 2021.

[MQS16] Milad Malekzadeh, Jeffrey Queißer, and Jochen J Steil. Learning the end-effector pose from demonstration for the bionic handling assistant robot. In *Proceedings of the International Workshop on Human Human Friendly Robotics*, 2016.

[MWG+21] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

[MYY+23] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, et al. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023.

[NAW+20] Michael Neunert, Abbas Abdolmaleki, Markus Wulfmeier, Thomas Lampe, Jost Tobias Springenberg, Roland Hafner, Francesco Romano, Jonas Buchli, Nicolas Heess, and Martin Riedmiller. Continuous-discrete reinforcement learning for hybrid control in robotics. *arXiv preprint arXiv:2001.00449*, 2020.

[NHR99] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[NZSL19] Suraj Nair, Yuke Zhu, Silvio Savarese, and Fei-Fei Li. Causal induction from visual observations for goal directed tasks. arXiv:1910.01751v1 [cs.LG], 2019.

[OAC18] Ian Osband, John Aslanides, and Albin Cassirer. Randomized prior functions for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 8617–8629, 2018.

[OBPVR16] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034, 2016.

[OWJ+22] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[PALvdP18] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 37(4):143, 2018.

[PAZA18] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3803–3810. IEEE, 2018.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Neural Information Processing Systems*, pages 8024–8035, 2019.

[PHD+17] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[PNWW23] Andrew Patterson, Samuel Neumann, Martha White, and Adam White. Empirical design in reinforcement learning. *arXiv preprint arXiv:2304.01315*, 2023.

[PQR+24] Abhishek Padalkar, Gabriel Quere, Antonin Raffin, João Silvério, and Freek Stulp. Guiding real-world reinforcement learning for in-contact manipulation tasks with shared control templates. *Autonomous Robots*, 48(4):12, 2024.

[PQS+23] Abhishek Padalkar, Gabriel Quere, Franz Steinmetz, Antonin Raffin, Matthias Nieuwenhuisen, João Silvério, and Freek Stulp. Guiding reinforcement learning with shared control templates. In *40th IEEE International Conference on Robotics and Automation, ICRA 2023*. IEEE, 2023.

[PS08] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[PS18] Aloïs Pourchot and Olivier Sigaud. Cem-rl: Combining evolutionary and gradient-based methods for policy search. *arXiv preprint arXiv:1810.01222*, 2018.

[PTLK17] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. *arXiv preprint arXiv:1712.00378*, 2017.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[QHI+20] Gabriel Quere, Annette Hagengruber, Maged Iskandar, Samuel Bustamante, Daniel Leidner, Freek Stulp, and Jörn Vogel. Shared control templates for assistive robotics. In *2020 IEEE international conference on robotics and automation (ICRA)*, pages 1956–1962. IEEE, 2020.

[RAB+23] Rajkumar Ramamurthy, Prithviraj Ammanabrolu, Kianté Brantley, Jack Hessel, Rafet Sifa, Christian Bauckhage, Hannaneh Hajishirzi, and Yejin Choi. Is reinforcement learning (not) for natural language processing: Benchmarks, baselines, and building blocks for natural language policy optimization. In *The Eleventh International Conference on Learning Representations*, 2023.

[Raf18] Antonin Raffin. RL Baselines Zoo. `https://github.com/araffin/rl-baselines-zoo`, 2018.

[Raf20] Antonin Raffin. RL Baselines3 Zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.

[Raf21] Antonin Raffin. Augmented auto-encoder training code for donkeycar simulator. `https://github.com/araffin/aae-train-donkeycar`, 2021.

[RBI06] Ludovic Righetti, Jonas Buchli, and Auke Jan Ijspeert. Dynamic hebbian learning in adaptive frequency oscillators. *Physica D: Nonlinear Phenomena*, 216(2):269–281, 2006.

[RDF16] Jens Reinecke, Bastian Deutschmann, and David Fehrenbach. A structurally flexible humanoid spine based on a tendon-driven elastic continuum. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4714–4721. IEEE, 2016.

[RDS21] Antonin Raffin, Bastian Deutschmann, and Freek Stulp. Fault-tolerant six-dof pose estimation for tendon-driven continuum mechanisms. *Frontiers in Robotics and AI*, 8:11, 2021.

[RFK+23] Tobias Rohrer, Lilli Frison, Lukas Kaupenjohann, Katrin Scharf, and Elke Hergenröther. Deep reinforcement learning for heat pump control. In *Science and Information Conference*, pages 459–471. Springer, 2023.

[RFS08] Thomas Rückstieß, Martin Felder, and Jürgen Schmidhuber. State-dependent exploration for policy gradient methods. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 234–249. Springer, 2008.

[RHE+20] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable Baselines3 contrib. `https://github.com/Stable-Baselines-Team/stable-baselines3-contrib`, 2020.

[RHG+21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[RHRH22] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

[RHT+19] Antonin Raffin, Ashley Hill, René Traoré, Timothée Lesort, Natalia Díaz-Rodríguez, and David Filliat. Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics. *arXiv preprint arXiv:1901.08651*, 2019.

[RI08] L. Righetti and A.J. Ijspeert. Pattern generators with sensory feedback for the control of quadruped locomotion. In *2008 IEEE International Conference on Robotics and Automation*, pages 819–824, Pasadena, USA, 2008. IEEE.

[Rie05] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*, pages 317–328. Springer, 2005.

[RKS21] Antonin Raffin, Jens Kober, and Freek Stulp. Smooth exploration for robotic reinforcement learning. In *Conference on Robot Learning*, 2021.

[RLTK17] Aravind Rajeswaran, Kendall Lowrey, Emanuel V. Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[RSK+22] Antonin Raffin, Daniel Seidel, Jens Kober, Alin Albu-Schäffer, João Silvério, and Freek Stulp. Learning to exploit elastic actuators for quadruped locomotion. *arXiv preprint arXiv:2209.07171*, 2022.

[RSK+24] Antonin Raffin, Olivier Sigaud, Jens Kober, Alin Albu-Schäffer, João Silvério, and Freek Stulp. An open-loop baseline for reinforcement learning locomotion tasks. *Reinforcement Learning Journal*, 1:92–107, 2024.

[RSS+10] Thomas Rückstiess, Frank Sehnke, Tom Schaul, Daan Wierstra, Yi Sun, and Jürgen Schmidhuber. Exploring parameter space in reinforcement learning. *Paladyn, Journal of Behavioral Robotics*, 1(1):14–24, 2010.

[RVR+17] Andrei A Rusu, Matej Večerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with

progressive nets. In *Conference on robot learning*, pages 262–270. PMLR, 2017.

[RW11] D Caleb Rucker and Robert J. Webster III. Statics and dynamics of continuum robots with general tendon routing and external loading. *IEEE Trans. on Robotics*, 27(6):1024–1030, 2011.

[SA19] Adam Stooke and Pieter Abbeel. rlpyt: A research code base for deep reinforcement learning in pytorch. arXiv: 1909.01500v2 [cs.LG], 2019.

[SB18a] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SB18b] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[Sen20] Takuma Seno. d3rlpy: A data-driven deep reinforcement learning library as an out-of-the-box tool. `https://github.com/takuseno/d3rlpy`, 2020.

[SGFH21] Jonah Siekmann, Yesh Godse, Alan Fern, and Jonathan Hurst. Sim-to-real learning of all common bipedal gaits via periodic reward composition. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7309–7315. IEEE, 2021.

[SGS+21] Tim Seyde, Igor Gilitschenski, Wilko Schwarting, Bartolomeo Stellato, Martin Riedmiller, Markus Wulfmeier, and Daniela Rus. Is bang-bang control all you need? solving continuous control with bernoulli policies. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[SHC+17] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[SHG+20] Daniel Seidel, Milan Hermann, Thomas Gumpert, Florian C. Loeffl, and Alin Albu-Schäffer. Using elastically actuated legged robots in rough terrain: Experiments with DLR quadruped bert. In *2020 IEEE Aerospace Conference*, pages 1–8, 2020.

[SHHR14] Freek Stulp, Laura Herlant, Antoine Hoarau, and Gennaro Raiola. Simultaneous on-line discovery and improvement of robotic skill options. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1408–1413. IEEE, 2014.

[SHI+23] Raghav Soni, Daniel Harnack, Hauke Isermann, Sotaro Fushimi, Shivesh Kumar, and Frank Kirchner. End-to-end reinforcement learning for torque based variable height hopping. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7531–7538. IEEE, 2023.

[SKL23] Laura Smith, Ilya Kostrikov, and Sergey Levine. Demonstrating a walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *Robotics: Science and Systems (RSS) Demo*, 2(3):4, 2023.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[SMC+17] Felipe Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 12 2017.

[SML+15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[SOR+10] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

[SOW+20] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[SPB22] Leon Sievers, Johannes Pitz, and Berthold Bäuml. Learning purely tactile in-hand manipulation with a torque-controlled hand. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 2745–2751. IEEE, 2022.

[SRM+23] Yunlong Song, Angel Romero, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Reaching the limit in autonomous racing: Optimal control versus reinforcement learning. *Science Robotics*, 8(82):eadg1462, 2023.

[SRSB12] Korbinian Schmid, Felix Ruess, Michael Suppa, and Darius Burschka. State estimation for highly dynamic flying systems using key frame odometry with varying time delays. In *International Conference on Intelligent Robots and Systems*, pages 2997–3004. IEEE, 2012.

[SS13] Freek Stulp and Olivier Sigaud. Robot skill learning: From reinforcement learning to evolution strategies. *Paladyn, Journal of Behavioral Robotics*, 4(1):49–61, 2013.

[SS15] Freek Stulp and Olivier Sigaud. Many regression algorithms, one unified model – a review. *Neural Networks*, 2015.

[SS19] Olivier Sigaud and Freek Stulp. Policy search in continuous action domains: an overview. *Neural Networks*, 2019.

[SS23] Thomas Simonini and Omar Sanseviero. The hugging face deep reinforcement learning class. `https://github.com/huggingface/deep-rl-class`, 2023.

[SSKS21] Yunlong Song, Mats Steinweg, Elia Kaufmann, and Davide Scaramuzza. Autonomous drone racing with deep reinforcement learning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1205–1212. IEEE, 2021.

[SSL+24] Daniel Seidel, Annika Schmidt, Xiaozhou Luo, Antonin Raffin, Luisa Mayershofer, Tristan Ehlert, Davide Calzolari, Milan Hermann, Thomas Gumpert, Florian Loeffl, Emiel Den Exter, Anne Köpken, Rute Luz, Adrian Simon Bauer, Nesrine Batti, Florian Samuel Lay, Ajithkumar Narayanan Manaparampil, Alin Albu-Schäffer, Daniel Leidner, Peter Schmaus, Thomas Krüger, and Neal Y. Lii. Toward space exploration on legs: Iss-to-earth teleoperation experiments with a quadruped robot. In *2024 IEEE Conference on Telepresence*, 2024.

[STA16] F. Steidle, A. Tobergte, and A. Albu-Schäffer. Optical-inertial tracking of an input device for real-time robot control. In *International Conference on Robotics and Automation*, pages 742–749. IEEE, 2016.

[Sto23] Manuel Stoiber. *Closing the Loop: 3D Object Tracking for Advanced Robotic Manipulation*. PhD thesis, Technische Universität München, 2023.

[STS12] Freek Stulp, Evangelos A Theodorou, and Stefan Schaal. Reinforcement learning with sequences of motion primitives for robust manipulation. *IEEE Transactions on robotics*, 28(6):1360–1370, 2012.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv:1707.06347v2 [cs.LG], 2017.

[SYH+19] Avi Singh, Larry Yang, Kristian Hartikainen, Chelsea Finn, and Sergey Levine. End-to-end robotic reinforcement learning without reward engineering. In *Robotics: Science and systems*, Freiburg im Breisgau, Germany, June 22-26 2019.

[TBKW20] Arne Traue, Gerrit Book, Wilhelm Kirchgässner, and Oliver Wallscheid. Toward a reinforcement learning environment toolbox for intelligent electric motor control. *IEEE Transactions on neural networks and learning systems*, 33(3):919–928, 2020.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[TFR+17] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[TTK+23] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, 2023.

[TZC+18] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[UO30] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[VDR+24] Francecso Vezzi, Jiatao Ding, Antonin Raffin, Jens Kober, and Cosimo Della Santina. Two-stage learning of highly dynamic motions with rigid and articulated soft quadrupeds. In *International Conference on Robotics and Automation, ICRA 2024*. IEEE, 2024.

[VGK+23] Francesco Maria Velotti, Brennan Goddard, Verena Kain, Rebecca Ramjiawan, Giovanni Zevi Della Porta, and Simon Hirlaender. Towards automatic setup of 18 mev electron beamline using machine learning. *Machine Learning: Science and Technology*, 4(2):025016, 2023.

[VH18] Anna-Lisa Vollmer and Nikolas J Hemion. A user study on robot skill learning without a cost function: Optimization of dynamic movement primitives via naive user feedback. *Frontiers in Robotics and AI*, 5:77, 2018.

[vHTP17] H. van Hoof, D. Tanneberg, and J. Peters. Generalized exploration in policy search. *Machine Learning*, 106(9-10):1705–1724, oct 2017. Special Issue of the ECML PKDD 2017 Journal Track.

[WEH+22] Philipp Wu, Alejandro Escontrela, Danijar Hafner, Ken Goldberg, and Pieter Abbeel. Daydreamer: World models for physical robot learning, 2022.

[WHB+23] Sebastian Wolf, Cynthia Hofmann, Thomas Bahls, Henry Maurenbrecher, and Benedikt Pleintinger. Modularity in humanoid robot design for flexibility in system structure and application. In *2023 IEEE-RAS 22nd International Conference on Humanoid Robots (Humanoids)*, pages 1–7. IEEE, 2023.

[Whi23] Peter Whidden. Playing pokemon red with reinforcement learning. `https://github.com/PWhiddy/PokemonRedExperiments`, 2023.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[WTGE20] Steven Wang, Sam Toyer, Adam Gleave, and Scott Emmons. The `imitation` library for imitation learning and inverse reinforcement learning. `https://github.com/HumanCompatibleAI/imitation`, 2020.

[WZD+20] Jiayi Weng, Minghao Zhang, Alexis Duburcq, Kaichao You, Dong Yan, Hang Su, and Jun Zhu. Tianshou. `https://github.com/thu-ml/tianshou`, 2020.

[YLS10] Gang Yu, Changning Li, and Jun Sun. Machine fault diagnosis based on gaussian mixture model and its application. *The International Journal of Advanced Manufacturing Technology*, 48(1-4):205–212, 2010.

[YZC+22] Yuxiang Yang, Tingnan Zhang, Erwin Coumans, Jie Tan, and Byron Boots. Fast and efficient locomotion via learned gait transitions. In *Conference on Robot Learning*, pages 773–783. PMLR, 2022.

[ZFW+23] Ziwen Zhuang, Zipeng Fu, Jianren Wang, Christopher G Atkeson, Sören Schwertfeger, Chelsea Finn, and Hang Zhao. Robot parkour learning. In *7th Annual Conference on Robot Learning*, 2023.

[ZQW20] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE symposium series on computational intelligence (SSCI)*, pages 737–744. IEEE, 2020.