Internship at DFKI: Automatic Subtask Discovery and Concurrent Learning of Subtasks for Stepping Locomotion using Hierarchical Reinforcement Learning

Matti SOUCAILLE
Master MVA 2024–2025, ENS Paris-Saclay, France
matti.soucaille@ens-paris-saclay.fr

September 12, 2025

Abstract

Hierarchical Reinforcement Learning (HRL) is an efficient framework for tackling challenging, long-horizon tasks by decomposing them into simpler subtasks, that will operate at different temporal scales. Notwithstanding its capacity to enhance sample efficiency and long-term credit assignment, one of the key bottleneck in HRL remains the autonomous discovery of these subtasks and the scalability of their learning. This challenge is especially pertinent in robotics, where long training times and limited hardware access make efficient learning necessary. This internship aims to address these limitations through three core research axes. First, we investigate whether HRL offers tangible benefits over standard RL approaches in the context of stepping locomotion. Second, we assess the implications of concurrent learning in contrast to sequential methods, hypothesizing that parallelization can significantly reduce training time while maintaining or improving performance. Third, we develop a subtask discovery pipeline based on trajectory segmentation using object perception models and large language models (LLMs), which enables automatic annotation and integration of subtasks within the Option framework. The proposed methods have been tested in simulation environments adapted for stepping locomotion and integrated with the ARTER robot platform at DFKI. First results show that HRL and parallelization effectively leads to faster convergence and improved modularity of learned behaviors, showing the relevance of this direction for scalable robotic learning. Overall, this work intends to fill the gap between hierarchical abstractions and practical, deployable reinforcement learning systems in robotics.

Contents

1 Introduction **5**

2 Context **6**
2.1 About the DFKI . . . 6
2.2 Presentation of the ARTER robot [1] . . . 6
2.3 Internship overview . . . 8

3 Background **9**
3.1 Reinforcement Learning . . . 9
3.1.1 Policy and Value Functions . . . 9
3.1.2 Limitations and challenges . . . 10
3.2 Hierarchical Reinforcement Learning . . . 10
3.2.1 Formal Definition of a Subtask . . . 10
3.2.2 Hierarchical Reinforcement Learning as a Semi-Markov Decision Process . . . 11
3.2.3 Problem Formulation in HRL . . . 12
3.2.4 Feudal Paradigm . . . 12
3.2.5 Policy Tree paradigm . . . 12
3.2.6 Challenge . . . 13

4 Related work **14**
4.1 Unified Subtask Discovery and Learning (UNI) . . . 14
4.2 Independent Subtask Discovery (ISD) . . . 15
4.3 Discovery with Foundation Model (DFM) . . . 15
4.4 Detail of important algorithm . . . 16
4.4.1 HIRO . . . 16
4.4.2 Option-Critic . . . 16

5 Motivation **18**
5.1 Limit of existing methods . . . 18
5.2 Problem statement . . . 19

6 Methodology **20**
6.1 ARTER Environment . . . 20
6.1.1 Action . . . 22
6.1.2 Observation . . . 23
6.1.3 Reward . . . 24
6.1.4 Termination . . . 25
6.2 Comparison of RL and HRL algorithm . . . 25
6.3 Comparison of sequential and parallelized training . . . 26
6.4 LLM Pipeline . . . 27
6.4.1 Save trajectory . . . 28
6.4.2 Segmentation . . . 28
6.4.3 Segmentation annotation . . . 29
6.4.4 Task reward . . . 30
6.4.5 Training . . . 30

7 Experimentation and evaluation 32
7.1 RL versus HRL . . . 32
7.2 Sequential versus parallelized . . . 34
7.3 LLM Pipeline . . . 35

8 Discussion 37

9 Future work 38

10 Conclusion 40

A Additional Figures 41
A.1 ARTER . . . 41
A.1.1 DOF . . . 41
A.1.2 Detailled controller scheme . . . 42
A.2 ARter environment detail . . . 42
A.2.1 Action space . . . 42
A.2.2 Observation space . . . 43
A.2.3 Wrapper . . . 45
A.3 Complete LLM pipeline result . . . 45

List of Figures

2.1 Kinematic of the Key Components of the ARTER Robot. Figure from Babu and all [1] . . 7

3.1 Illustration of Feudal RL the observation and reward of higher manager can happened at different temporal scale enabling the temporal abstraction of HRL 12
3.2 Illustration of policy Tree in Hierarchical Reinforcement Learning the observation and reward of higher managers can happened at different temporal scale enabling the temporal abstraction of HRL 13

4.1 Illustration of the HIRO architecture. Figure from [2]. 17
4.2 Illustration of the option-critic architecture. Figure from [3] 17

6.1 Image from the simulation of ARTER, the ground and the obstacle 20

7.1 Comparison of algorithms across environments: time and reward 32
7.2 Comparison of four RL algorithms. Each subplot shows the evolution of the coefficient of each reward 33
7.3 Impact of parallelization on time to reach in the ARTER environment, with variance shaded. 34
7.4 Example of the segmentation of a video 35

A.1 Details of the mid-level controller structure. Figure from [1] 42

List of Tables

2.1 Summary of ARTER robot design and architecture 7

6.1 Comparison of data structuring strategies in TorchRL pipeline 22

7.1 Qualitative comparison of interpretability and transferability across RL algorithms 33
7.2 Evaluation of Algorithm Speeds: Time to Reach 1,000,000 Steps (in Seconds) 34
7.3 Evaluating PPO Performance in ARTER: Impact of Parallel Environment Scaling on Reward Optimization for a training of 3600 seconds. 35
7.4 Comparison of Precision, Recall, and F1 Score for Temporal Segmentation Algorithms for 10 different videos 36
7.5 Segment Details Referenced by Node Number 36

A.1 Summary of ARTER Robot Degrees of Freedom 41
A.2 Summary of the action of the ARTER env 43
A.3 Summary of the observation of the ARTER env 44
A.4 Recap of the different wrappers implemented in the environment 45
A.5 Segment Details Referenced by Node Number 47

Chapter 1

Introduction

Hierarchical Reinforcement Learning is a novel framework that achieves very promising results to address complex or long-horizon problems. The key idea is to decompose a complex global task into a hierarchy of simpler subtasks, that an algorithm can more easily learn. The hierarchy creates a temporal abstraction where the high-level tasks are updated fewer times than lower ones. This temporal abstraction greatly improves the efficiency of exploration, the credit assignment, and finally the learning of the policy. Thus HRL is particularly appealing in the context of robotics and especially locomotion. Despite the theoretical promise of HRL, in practice using HRL means solving two problems: finding useful subgoals or subtasks and learning efficiently those subtasks ideally in parallel.

This internship, conducted at the German Research Center for Artificial Intelligence (DFKI), addresses the challenge in the context of stepping locomotion for a wheel-legged robot. The project, titled *Automatic subtask discovery and concurrent learning of subtasks for stepping locomotion using hierarchical reinforcement learning*, will be organized around three core research axes:

1. **Evaluating HRL Advantages Over Standard RL:** We will evaluate to what extent HRL effectively improves the learning in long-horizon problems compared to RL.

2. **Parallelized vs. Sequential Subtask Learning:** We will try to evaluate the potential speed gain of parallelized learning and to what extent it can affect the overall learning.

3. **Trajectory Segmentation and Reward Design Using the Option Framework:** Finally, using the results of the two previous axes, we will develop a pipeline for the automatic segmentation of agent trajectories into meaningful subtasks. These segments are annotated and integrated into the HRL pipeline using the Option framework to define subtask rewards and termination conditions in a minimally supervised manner.

These axes contribute to the development of scalable and autonomous learning in HRL. In particular, the proposed trajectory segmentation pipeline, leveraging large pre-trained models aims to reduce human intervention in defining subgoals and task structure.

The project is embedded in a real-world robotic context: the ARTER robot, a wheel-legged robotic platform developed at DFKI for excavation and navigation in rough environments. This applied setting provides a unique testbed for evaluating the effectiveness of HRL in realistic, unstructured domains. By integrating subtask discovery and parallelized learning in this platform, the project seeks to bridge the gap between theoretical HRL frameworks and deployable, adaptive robotic systems.

Chapter 2

Context

2.1 About the DFKI

An overview of the German Research Center for Artificial Intelligence (DFKI)

Founded in 1988 as a public-private partnership with headquarter in Kaiserslautern, the German Research Center for Artificial Intelligence (DFKI) has grown into one of Europe's leading institutions for AI research through its several branches in Bremen, Saarbruckern and Berlin. Its hundred of researchers, engineers and doctoral candidates aims to fill the gap between academia, industry, and government, encouraging innovation in areas such as natural language processing, robotics, embedded systems, and machine learning, through international collaborations and partnerships with universities, companies, and startups.

Focus on robotics at DFKI

The Robotics Innovation Center (RIC) is based in Bremen and is one of the major hub for the development of intelligent robotic systems in Germany. The different teams of the center aims to design robots that can navigate and perform tasks in unpredictable, dynamic, and often hazardous environments from a wide spectrum from deep-sea exploration and space missions to urban search and rescue and industrial automation. DFKI's robotics teams integrate the whole chain of design of a robot from mechanical design and sensor systems to motion planning, control strategies, and AI-driven learning algorithms in order to build robust platforms built for real-world deployment.

Robotic Terrestrial Team

In the RIC, the Robotic Terrestrial Team led by Ajish Babu, focuses on mobile robotic systems in terrestrial environments. Excavation, decontamination, and emergency response in areas that are inaccessible or unsafe for humans are part of the scenarios tackle by the team where robotics can provide promising solution. One of the team main goal is the development of the ARTER robot, a wheel-legged excavator designed to combine mobility and functionality in rugged, uneven terrain.

2.2 Presentation of the ARTER robot [1]

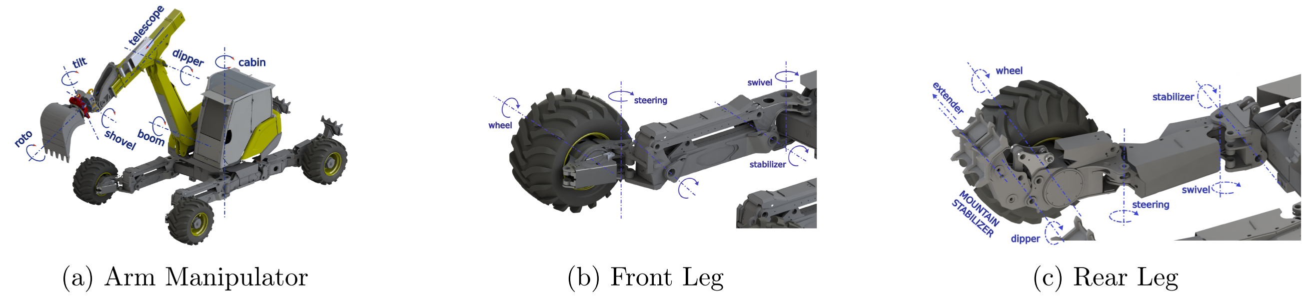
The ARTER system (Autonomous Robotic TEleoperation and Rescue platform) is a robotic platform developed by the German Research Center for Artificial Intelligence (DFKI) as part of the ROBDEKON competence center. ARTER combines a robust mechanical design and an autonomous controller to navigate and operate in complex scenarios. As summarized in Table 2.1, the platform is built upon the Menzi-Muck M545 base, a versatile walking excavator of 13 tons and 27 degrees of freedom (DOF). ARTER is known for its adaptability in uneven terrain thanks to its configuration : a manipulator arm and four articulated legs, two front and two rear. These different components have to work together in order to allow ARTER to move in complex landscapes.

The 27 degrees of freedom (DoF) of the robot are distributed across revolute and prismatic joints. All joints are hydraulically actuated and in our context will all operate under position and velocity control

Base Platform	Menzi-Muck M545
Total Mass	Approx. 13 tons
Degrees of Freedom (DoF)	27
Main Components	1 manipulator arm - 2 front legs - 2 rear legs

Table 2.1: Summary of ARTER robot design and architecture

modes, with corresponding sensor feedback for precise motion execution. The manipulator is made of seven joints whereas each legs incorporates six joints (see table A.1 for details), as previously explained this configuration aims to enable an adaptive locomotion and stabilization.



(a) Arm Manipulator (b) Front Leg (c) Rear Leg

Figure 2.1: Kinematic of the Key Components of the ARTER Robot. Figure from Babu and all [1]

Control Architecture: ARTER-MCS

One of ARTER's most unique features is its ability to combine manipulator and leg-based locomotion, enabling it a stepping mode : The manipulator lifts wheels off the ground (typically two at a time) which, enables the robot to step over obstacles or traverse extreme terrain.

To manage the robot's complex capabilities, DFKI has developed a custom Motion Control System (ARTER-MCS) responsible for remote control and autonomous operation. His design includes three structures low-level, mid-level, and high-level layers. The details about each structure is given in order to better understand the complexity of the stepping locomotion task and the use of RL.

The top layer of the ARTER Motion Control System works like a manager. It connects operators or mission control to the robot's internal systems. This layer handles instructions, sensor data, and information about the surroundings. Its job is to turn big-picture goals into specific actions the robot can carry out.

Embedded within this layer are specialized modules such as the footprint changer, which choreographs joints sequences to dynamically reshape the robot's stance, and the manipulator planner, which governs dexterous interactions within the environment. A Simultaneous Localization and Mapping (SLAM) module further enriches spatial awareness, constructing real-time maps that inform navigation and task execution. These components collectively supply the mid-level controllers enabling context-aware decision-making and precise locomotion across complex terrains.

The mid-level control layer of ARTER-MCS details in A.1 acts as a bridge between overall planning and actual robot movement. It takes high-level instructions, like walking, adjusting posture, or using robotic arms; and converts them into precise joint-level actions. This is done using tools like inverse kinematics solvers and servoing algorithms, which ensure smooth and accurate execution.

The mid-level layer also houses terrain adaptation and stepping controllers, which use reinforcement learning to dynamically adjust suspension parameters and gait patterns in response to uneven or unpredictable ground conditions. These adaptive mechanisms ensure robust mobility and environmental responsiveness. Additionally the Normalized Energy Stability Margin (NESM) informs the autonomous systems and the human operators of risk of tip-over, to ensure safe and reliable performance.

At the base of ARTER's control system is the low-level control layer, which acts like a hardworking

assistant carrying out detailed instructions from the higher layers. Its main job is to control all the robot's joints and movements. However, it has to deal with complex mechanical setups, sensors placed in different ways, various types of control methods, and the unpredictable behavior of hydraulic systems. This layer ensures that, despite the previous challenges, the robot moves smoothly and accurately.

Each joint may operate within distinct control domains, sensor space, actuator space, or joint space, necessitating continuous kinematic transformations to maintain command fidelity. Mechanical constraints often preclude direct sensor alignment with joint axes, further complicating feedback interpretation. Additionally the nonlinearities introduced by hydraulic valve can not be handle by classical PID controllers thus the use of lookup-table-based linearization techniques to achieve effective effort control.

2.3 Internship overview

This internship contributes to the development of a component within the mid-level controller module of the ARTER robot, specifically focused on stepping locomotion through reinforcement learning (RL). In the robot's hierarchical control architecture, the high-level controller plans the global trajectory, which the mid-level controller translates into joint-level commands, either in terms of velocity or position, executed by the low-level controller.

The mid-level controller of the ARTER robot aims to rely on an RL-based stepping mechanism in order to handle local obstacles along the planned path. As previously explained to achieve a more adaptive and high-level movement, the RL controller adjusts individual Degrees of Freedom (DoFs) via the low-level controller, rather than interfacing directly with the robot's hydraulic systems.

The previous work made by Ajish Babu in [1] has introduced a hybrid reinforcement learning (HRL) approach in order to train the ARTER robot for stepping motions. To improve ARTER's locomotion, the study pinpointed five key upcoming directions :

- **Transferring Solutions to Real System:** Transferring terrain adaptation and stepping locomotion from simulation to the real ARTER robot is complex due to a simulation-reality gap and the system's non-linear dynamics. Rather than fully modeling these dynamics, matching joint-level controller behavior may suffice. Realistic terrain generation domain randomization of terrain properties could help train robust controllers.

- **Safety and Robustness of DRL Controllers:** In order to guarantee safety and robustness with Deep Reinforcement Learning (DRL) algorithm, we must verify that the algorithm respect all needed objectives and constraints and allow human intervention as can do linear and some non-linear systems.

- **Automatic Subtask Discovery:** Designing subtask hierarchies manually in Hiearchical Deep Reinforcement Learning (HDRL) can be a hard manual work. Automatic discovery methods seek to learn the task structure and sub-goals. Though scalability and generalization remain active research areas, these methods improve learning efficiency and reusability without manual engineering.

- **Simultaneous Training of Subtasks:** Currently, subtasks are trained sequentially, but concurrent learning could boost efficiency and allow experience sharing. However, non-stationary environments and changing low-level policies can destabilize higher-level learning. Addressing this could enhance hierarchical reinforcement learning's stability and scalability.

- **Lifelong Learning:** The goal of lifelong learning is to allow the robot to continuously, even in production, adapt and develop new skills. Therefore, the need of intelligent and resilient robots capable of operating in dynamic environments can be addressed thanks to progress in incremental learning, skill transfer, and autonomous exploration.

This internship focuses on investigating two of these directions: automatic subtask discovery and simultaneous training of subtasks. This work seeks to evaluate the feasibility of HRL in combination with parallelized simulation environments to enhance training efficiency and controller generalization.

Chapter 3

Background

3.1 Reinforcement Learning

According to Sutton and Barto [4], Reinforcement Learning (RL) aims to compute the best possible action whatever the current configuration of the environment or state, for an agent in a given environment. This mapping of an action to each state is called a policy, so RL aims to find the best possible policy in a given environment.

Sutton and Barto categorize RL methods into two primary classes. *Model-free* approaches, such as Q-learning and SARSA, estimate value functions directly from experience without constructing an explicit model of the environment. In contrast, *model-based* methods involve learning a dynamic representation of the environment, which can then be used for planning and policy optimization.

Finally, they highlight a key concept in RL the : *exploration-exploitation trade-off*, which requires agents to balance the discovery of new actions (exploration) with the utilization of known strategies that yield high rewards (exploitation).

RL is typically formalized using a Markov Decision Process (MDP), defined by the tuple $\langle S, A, P, r \rangle$:

- S: Set of states representing the environment.
- A: Set of actions available to the agent.
- $P(s' \mid s, a)$: Transition probability function, describing the likelihood of moving to state s' after taking action a in state s.
- $r(s, a)$: Reward function, giving feedback for taking action a in state s.

An MDP is **stochastic** if there exists a state-action pair (s, a) such that $P(s' \mid s, a) < 1$, and **deterministic** if their exist an action a and a state s' such that $P(s' \mid s, a) = 1$ for all state s of the environment.

3.1.1 Policy and Value Functions

An agent follows a policy $\pi : S \times A \to [0, 1]$, which may be stochastic. The expected cumulative reward of taking action a_t in state s_t under policy π is given by the Q-value function:

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_{a \sim \pi(s)} \left[\sum_{i=1}^{\infty} \gamma^i r(s_{t+i}, a_{t+i}) \mid s_t, a_t \right] \quad (3.1)$$

where:

- $\gamma \in [0, 1)$ is the discount factor.
- $a \sim \pi(s)$ denotes sampling actions from the policy.

The goal of RL is to learn an optimal policy π^* that maximizes the Q-value for all state-action pairs:

$$\pi^* = \arg\max_{\pi} Q^{\pi}(s, a), \quad \forall s \in S, \forall a \in A \tag{3.2}$$

Agents learn by collecting experience samples (s_t, a_t, r_t, s_{t+1}), which form trajectories that capture the stochastic nature of both the policy and the environment.

3.1.2 Limitations and challenges

Despite its capabilities and application [5], RL faces several challenges. The mains limits come in large or continuous state spaces where RL struggles with scalability. Moreover sparse or delayed rewards make learning inefficient. Finally, some tasks demand reasoning over long-term dependencies, which standard RL methods may not handle well because of its lacks of abstraction, treating all decisions at the same level of granularity.

In such settings, not only exploration becomes highly inefficient but credit assignment becomes unreliable, and policies tend to overfit or converge to suboptimal behaviors. These limitations make it difficult to learn complex behaviors end-to-end using standard approaches, motivating the need of Hierarchical Reinforcement Learning (HRL). Indeed HRL aims to overcome this limits by introducing temporal abstraction thanks to a decomposition of tasks into subtasks or options. This structure allows the system to learn useful skills that can be reused across different situations. Moreover the decision-making at multiple levels aims to improve the learning process and help the system adapt better to new scenarios.

3.2 Hierarchical Reinforcement Learning

When the state and action spaces are large and the task horizon is long, exploration and learning become increasingly harder for flat RL methods [6]. Hierarchical Reinforcement Learning (HRL) tackles these challenges thanks to abstraction by dividing one complex task into simpler subtasks. This division creates a hierarchy of policies where the **high-level policy** selects subtasks as its actions and the **low-level policy** combines the decision of the **high-level policy** with the observation of the environment to choose actions in the environment. In other words, the high-level policy is trained to solve the main task by choosing the best subtasks for each state, using the external rewards provided by the environment. Each **lower-level policy** is responsible for executing a selected subtask. These policies are themselves trained via reinforcement learning, using internal rewards specific to the subtask. Optionally, the main task reward may be incorporated to guide learning. At the bottom of the hierarchy, the **lowest-level policies** choose *primitive actions*, which directly interact with the environment. In the end, this hierarchical structure is supposed to enable more efficient exploration, reuse of learned behaviors, and improved scalability in complex environments.

3.2.1 Formal Definition of a Subtask

We adopt the definition and notation of subtasks as presented in "Hierarchical Reinforcement Learning: A Comprehensive Survey" by Pateria et al. [6] which are briefly reminded here. Let Γ denote the main long-horizon task, governed by the top-level policy π_Γ. Each subtask, denoted by ω, is characterized by the following components:

- **Policy** (π_ω): A mapping from environment states to either primitive actions or lower-level subtasks within ω.
- **Objective Components:**
  - r_ω: A reward signal specific to the subtask, used to train π_ω; typically distinct from the main task reward.
  - g_ω: A subgoal or set of subgoals associated with ω, which may be a concrete state $s \in S$, an abstract representation, or a learned embedding.
- **Execution Components:**

- I_ω: The initiation condition, specifying when ω can be activated. This may be defined as a set of states, a probabilistic function, or logical rules.
- β_ω: The termination condition, determining when ω concludes. The termination condition can be related to state like a set of terminal states or an association to each state of a termination probability, but it can also be a time constraint. If g_ω is defined, then if the agent reaches the subgoal, it triggers termination.

3.2.2 Hierarchical Reinforcement Learning as a Semi-Markov Decision Process

Hierarchical Reinforcement Learning (HRL) can be formalized using Semi-Markov Decision Processes (SMDPs) [7]. In contrast to standard Markov Decision Processes (MDPs), SMDPs explicitly model the temporal duration of actions. Therefore there are well-suited to represent subtasks in HRL.

Following the formalism introduce in [6], the agent operates over a hierarchy where each action corresponds to a subtask. Let the environment state at time t be $s_t \in S$, and let the agent select a subtask $\omega_t \in \Omega$, where Ω is the space of available subtasks. The execution of ω_t spans c_{ω_t} timesteps, determined by its termination condition β_{ω_t}. The transition dynamics of the SMDP are given by:

$$P(s_{t+c_{\omega_t}}, c_{\omega_t} \mid s_t, \omega_t) = P(s_{t+c_{\omega_t}} \mid s_t, \omega_t, c_{\omega_t}) \cdot P(c_{\omega_t} \mid s_t, \omega_t) \tag{3.3}$$

The cumulative reward obtained by executing subtask ω_t from state s_t is:

$$R(s_t, \omega_t) = \mathbb{E}_{a \sim \pi_{\omega_t}(s)} \left[\sum_{i=0}^{c_{\omega_t}-1} \gamma^i r(s_{t+i}, a_{t+i}) \mid s_t, a_t \right] \tag{3.4}$$

This reward reflects the expected return from following the subtask policy π_{ω_t} until termination.

It is possible to define recursively the optimal value function for the hierarchical policy as:

$$Q(s_t, \omega_t) = R(s_t, \omega_t) + \sum_{s_{t+c_{\omega_t}}, c_{\omega_t}} \gamma^{c_{\omega_t}} P(s_{t+c_{\omega_t}}, c_{\omega_t} \mid s_t, \omega_t) \max_{\omega'} Q(s_{t+c_{\omega_t}}, \omega') \tag{3.5}$$

Thanks to this formulation, we can highlight both the immediate reward from executing ω_t and the future value of subsequent subtasks.

Multi-Level Hierarchical Structure

To generalize HRL across multiple levels of abstraction, we define the following notation:

- ω_l: subtask at hierarchy level l
- Ω_{ω_l}: set of subtasks available under ω_l, such that $\omega_{l-1} \in \Omega_{\omega_l}$
- $\pi_{\omega_l} : S \times \Omega_{\omega_l} \to [0, 1]$: policy at level l mapping states to subtasks. Probability to select a certain subtask of Ω_{ω_l} in a given state.
- Ω_{ω_1} : primitive action space at the lowest level
- π_Γ and Ω_Γ: top-level policy and its subtask space for the main task Γ

The complete hierarchical policy is denoted by $\pi_{\text{hierarchy}}$, which ultimately maps states to primitive actions: $a = \pi_{\text{hierarchy}}(s)$.

The expected cumulative discounted reward under this hierarchical policy is:

$$Q_{\text{hierarchy}}(s_t, a_t) = \mathbb{E}_{a \sim \pi_{\text{hierarchy}} | \Omega_{\text{hierarchy}}} \left[\sum_{i=0}^{\infty} \gamma^i r(s_{t+i}, a_{t+i}) \mid s_t, a_t \right] \tag{3.6}$$

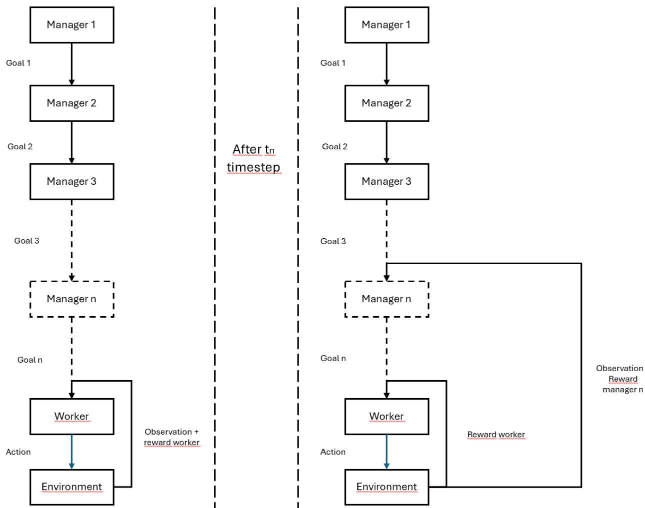


Figure 3.1: Illustration of Feudal RL the observation and reward of higher manager can happened at different temporal scale enabling the temporal abstraction of HRL

3.2.3 Problem Formulation in HRL

The central objective in HRL is to jointly discover the optimal hierarchical policy and the optimal subtask structure:

$$\Omega^*_{\text{hierarchy}}, \pi^*_{\text{hierarchy}} = \arg\max_{\Omega_{\text{hierarchy}}} \arg\max_{\pi_{\text{hierarchy}}|\Omega_{\text{hierarchy}}} Q_{\text{hierarchy}}(s,a), \quad \forall s \in S, a \in A \tag{3.7}$$

This problem can be decomposed into two interdependent components:

- **Hierarchical Policy Learning:** Learning $\pi^*_{\text{hierarchy}}$ conditioned on a given subtask structure $\Omega_{\text{hierarchy}}$. This can be approached either end-to-end or via bottom-up methods, where lower-level policies are learned first.
- **Subtask Discovery:** Inferring $\Omega^*_{\text{hierarchy}}$ from interaction data. While manual design is possible, automatic discovery is crucial for scalable and generalizable HRL. This last part will be the core of this internship.

One can identify two structural paradigms within HRL : the *feudal hierarchy* and the *policy tree.*

3.2.4 Feudal Paradigm

In the **feudal paradigm**, illustrates in 3.1, the higher-level policy (the "managers") assigns subgoals to a lower-level policy (other "managers" or "worker" for the lowest policies), which then can attempt to achieve the goal either by interacting directly with the environment or by assigning another goal to another lower-level therefore creating a hierarchy. The workers disposed of a universal policy and a goal given by the manager above him. It is important to note that the worker and managers can have different reward and act at different timescale (the worker can act at every step while the manager can act every 10 steps for example in order to avoid changing the worker goal too quickly).

3.2.5 Policy Tree paradigm

In contrast, the **policy tree paradigm**, illustrated in Figure 3.2 involves the higher-level policy directly selecting among distinct lower-level subtask policies, where *each subtask has its own dedicated policy*, rather than relying on a single universal one. Notable examples include the **Options framework**.

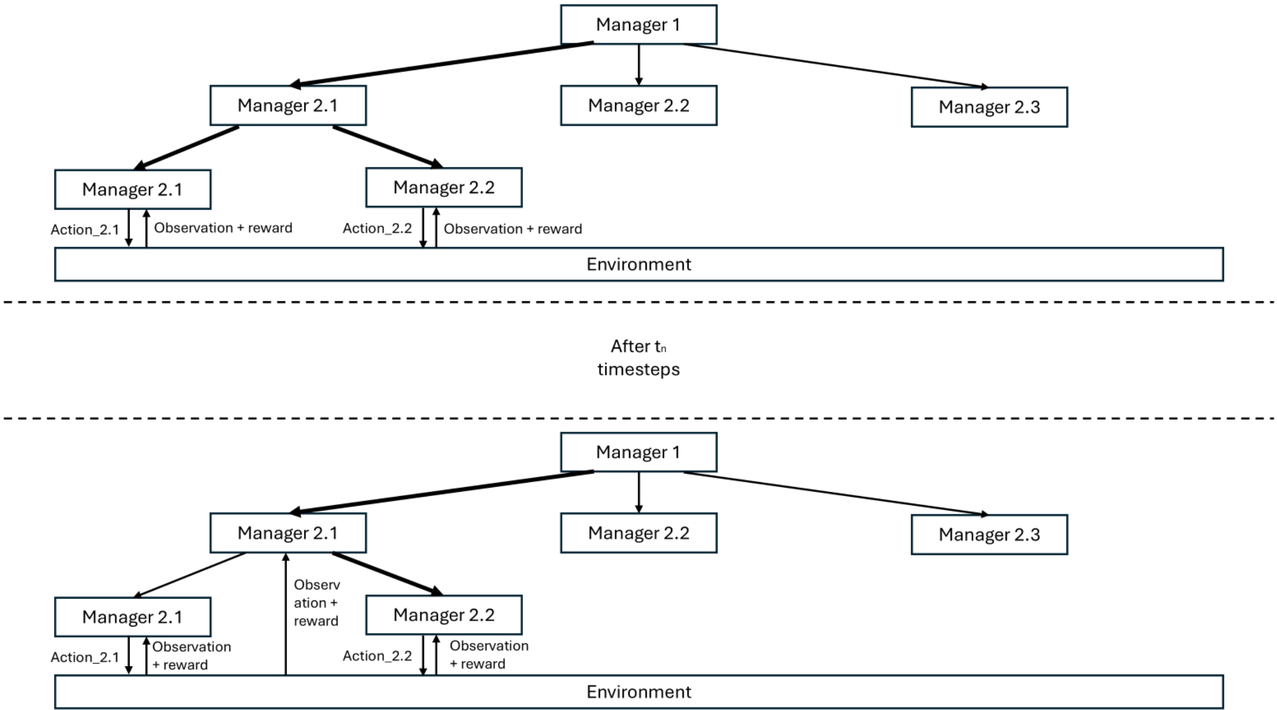


Figure 3.2: Illustration of policy Tree in Hierarchical Reinforcement Learning the observation and reward of higher managers can happened at different temporal scale enabling the temporal abstraction of HRL

The option Framework

A central paradigm for Policy-Tree is the **options framework**. It formalizes temporal abstraction in RL by introducing *options*, i.e., temporally extended actions. Each option ω is defined by:

- an initiation set I_ω (states where ω can be started),
- an intra-option policy π_ω (mapping states to actions or lower-level subtasks),
- a termination condition β_ω (determining when the option ends).

This enables an agent to plan and act at multiple time scales by composing primitive actions into higher-level skills. Options can represent anything from atomic motor commands to sophisticated skills such as "grasping an object".

The option-framework is the base of many algorithm under the Option-Critics architecture [3] which led to the development of different algorithm aims to improve the capacities of the Option-Critics architecture (Proximal policy option critic [8], hierarchical option critic [9], interest option critic [10]). But other algorithm in the context of the more general option framework has been developed and show promising results like Double Actor Critic [11].

3.2.6 Challenge

The success of HRL, however, hinges on the availability of high-quality subtask definitions. In the literature, these are typically manually specified by experts using domain knowledge, as learning expressive and effective goal spaces remains difficult and the automatic definition of useful subgoals is still unresolved in general context even if new algorithms begin to emerge.

Credit assignments also remains a challenge as it is hard to disentangle errors between high-level planning and low-level execution, and the simultaneous learning of the high level and low-level training of hierarchical policies is sensitive and often unstable. Therefore it can lead to redundant exploration and slow learning. Finally the lack of standardized benchmarks with clear hierarchical structure makes it hard to discover new efficient algorithm adapted to all kind of environment.

Chapter 4

Related work

This chapter provides a concise conceptual overview of the reinforcement learning methods and hierarchical extensions relevant to subtask discovery.

[6] proposes a taxonomy of Hierarchical Reinforcement Learning (HRL) algorithms, categorized into 6 main classes: TransferHRL, Learning Hierarchical Policy (LHP), UNIfied Subtask Discovery and Learning (UNI), Independant SUbtask Discovery (ISD) and Multi-Agent HRL (MAHRL). As our contribution focus mainly on single agent subtask discovery, we will focus from the previous classes only on UNI and ISD. More recently [12] introduces a new categories that is deeply link with our work : Discovery with Foundation Models (DFM).

4.1 Unified Subtask Discovery and Learning (UNI)

UNI algorithms *jointly perform subtask discovery and hierarchical policy learning* for a specific task [6]. This end-to-end approach reuses the same trajectory data for both segmentation of behaviors into subtasks and learning policies to solve them. This paradigm fits closely with the project's goal of minimizing human intervention in defining task structures. UNI methods enable automatic segmentation of behaviors into meaningful subtasks while simultaneously learning how to execute them. This is really helpful in unstructured environments, where manually specifying subgoals is impractical. What seem to be the most viable strategy explored in the UNI framework is the Option Framework detailled in 3.2.5.

One notable approach is the one proposed by Daniel et al [13] where the agent learns a set of simpler sub-policies as well as the initiation and termination probabilities for each of those sub-policies by inferring all relevant components from data. These options are learned globally from task rewards, but the number of options must be predefined, and they are not available during the initial learning phase. Similarly, **Skill Chaining** by Konidaris et al. [14] constructs a chain of options by identifying initiation states via classifiers and setting them as subgoals for other options. This method enables the flexible learning of subgoal-seeking options from scratch but is limited to goal-directed tasks.

One of the most promising method is the **Option-Critic** architecture introduced by Bacon et al. [3], which learns options using policy gradients derived from task rewards. Extensions such as **Proximal Policy Option-Critic** [8] and **Hierarchical Option-Critic** [9] adapt the framework to continuous action spaces and multi-level hierarchies, respectively. Despite their generality, these methods often require a predefined number of options and may struggle with sparse reward environments. The **Interest Option-Critic** [10] introduces an interest function to learn initiation conditions, while **Double Actor-Critic** [11] leverages standard policy optimization algorithms to learn options, though it risks degenerating into frequently terminating behaviors.

Feudal HRL approaches offer an alternative by structuring policies into manager-worker hierarchies. **HIRO** [2] reduces non-stationarity by learning subgoal policies at fixed timestep or specific states, though mapping of state to subgoal space can be challenging in high-dimensional environments. **FuN** [15] and **Sub-optimality Bound HRL** [16] aim to learn low-dimensional subgoal spaces using policy gradients and theoretical guarantees, respectively. While these different methods can provide structured learning and improved sample efficiency, they will lack the expressiveness needed for behaviors where subgoal are hard to defined such as continuous motion or navigation in dynamic environments.

4.2 Independent Subtask Discovery (ISD)

These algorithms focus on the *automatic discovery of subtasks*, such as subgoals or bottleneck states, from experience data or interaction logs. This discovery typically occurs during a *pre-training phase* and tries to find *task-agnostic* and *transferable* subtasks.

ISD methods are instrumental to create a set of potential reusable behaviors in the context of robotic locomotion. In this context identifying bottleneck states, such as transitions between stable footholds or terrain types, can improve sample efficiency and generalization. Moreover ISD allow to train concurrently without strict temporal dependencies. As in complex robotic systems like ARTER, different locomotion strategies have to be learned and deployed dynamically, the modularity of ISD is key.

According to [6] ISD methods can be broadly categorized into two mains approaches : bottleneck-based and latent skill embedding. Finding important states that frequently appear across successful trajectories or lie at the boundaries of state-space partitions is the core idea of Bottleneck-based approach. For instance, McGovern et al. [17] and Şimşek et al. [18] propose to use frequently visited states as subgoals. These methods seem simple but requires extensive exploration to gather sufficient trajectory data. Graph-based techniques such as Q-cut from Menache et al. [19] and L-cut from Şimşek et al. [20] are based on graph-cut algorithms and aim to partition the state space and identify bottlenecks. They are very efficient reducing the need of data but are still limited in scalability in continuous or high-dimensional environments. Machado et al. [21] introduce the use of Proto-Value Functions (PVFs) to locate subgoals at local maxima, which generalizes well across tasks but can be computationally expensive. Similarly, HASSLE from Bakker et al. [22] clusters the state space, and uses centroids as subgoals. However these methods struggles with scalability in high-dimensional domains.

Latent skill embedding methods take a more implicit approach by learning continuous representations of subgoals or skills from interaction data. HSP from Sukhbaatar et al. [23] with asymmetric self-play discovers a continuous subgoal embedding, enabling diverse behavior generation but do not generalize to tasks which doesn't have a clear subgoal structure. VALOR from Achiam et al. Both Hausman et al. [24] and [25] discover policies that minimize cross-entropy between latent skill variables and trajectories or maximize mutual information. However even if these approaches are successful in identifying a variety of reusable skills, they still need a lot of trials and errors and careful adjustments. SecTAR from Co-Reyes and associates [26] goes one step further in this paradigm by decoding latent embeddings of trajectories associated with particular skills, albeit at the expense of high data demands. In general, different methods balance data efficiency, scalability, and interpretability. However, their suitability depends on the complexity of the environment and how precise the subtasks need to be.

4.3 Discovery with Foundation Model (DFM)

Agents that learn skills through direct interaction with their environments face significant complexity because they require to organize raw experience into meaningful structures. Recent approaches leverage large pretrained models (LLMs) that encode prior knowledge and offer interpretable, language-based abstractions. These models facilitate HRL by enabling task decomposition and goal specification in open-ended domains. Several methods have emerged: using embeddings for reward shaping, employing LLMs for feedback, generating code to define reward functions, and treating LLMs as goal-conditioned policies. While not all approaches are inherently hierarchical, they collectively advance HRL by transforming latent linguistic knowledge into actionable skills. [12] made a classification of four main techniques : Providing Feedback, Reward as Code, Embedding Similarity and Directly Modeling the Policy. The last technique uses a large language model (LLM) to generate policy code for controlling the environment. In theory, this provides full interpretability of the policy. However, since it falls outside the scope of reinforcement learning (RL) and hierarchical RL (HRL), we will not go into details. Although these techniques haven't been adapted to HRL, they still offer strong inspiration for our work.

Providing Feedback Large language models (LLMs) provide a strong alternative to reward functions. By using their auto-regressive structure, they support chain-of-thought reasoning and in-context learning to generate feedback or preferences over states and trajectories. This enables more flexible reward modeling, using either binary success signals or preference comparisons turned into dense rewards. These

methods help exploration by capturing key behavioral milestones, improve credit assignment with consistent feedback over time. As shown by Bai et al. [27], this leads to agents with adaptive behavior across domains and simplifies reward learning without loss in performance or generalization.

Reward as code Xie et al. developped Text2Reward [28], leveraging the code generation capabilities of large language models (LLMs). LLMs offer a strong alternative to behavior evaluation by automatically creating reward functions from goal descriptions and symbolic environment data. This lets them generate executable code for task-specific rewards, combining domain knowledge, low-level features, and human feedback. It is very effective and scalable since it does not require querying during training and does not require learning parametric reward models. While applications in domains like Minecraft [29] demonstrate its efficacy for long-horizon goals, frameworks such as EUREKA from Ma et al. [30] show superhuman-level reward design across a variety of robotics tasks.

Embedding similarity In 2021, Radford et al. [31] developed a novel method call CLIP, that achieve promising result and have been derive in numerous algorithm. The method offer rich embedding spaces that enable agents to interpret and pursue language or image-based goals through cosine similarity-based reward functions. These embeddings, derived from contrastive pretraining, allow for goal-conditioned policies that learn complex behaviors in open-ended environments like Minecraft and robotics. The dense and interpretable nature of these rewards enhances exploration efficiency and facilitates transfer across diverse tasks, leveraging the compositional power of language. Moreover, methods using pretrained embeddings, whether fine-tuned, zero-shot, or trajectory-based, demonstrate strong generalization and sample efficiency, opening new avenues for scalable and instruction-following agents. This paradigm not only reduces the need for handcrafted rewards but also enables flexible curriculum design and robust skill acquisition.

4.4 Detail of important algorithm

This section aims to give a deeper insight of important algorithm in HRL that serve as a base for many other and will be use as a ground comparison in this work.

4.4.1 HIRO

Why It Matters. In Feudal RL, HIRO (HIerarchical Reinforcement learning with Off-policy correction) [2] is one of the main algorithm by addressing a fundamental challenge in HRL: how to train hierarchical policies efficiently using off-policy data. In HRL the instability of the low-level policy prevent the high-level policy to learn from collected experience. HIRO solved this problem by retroactively adjusts the high-level goals so that they match the behavior of the evolving low-level policy. HIRO was made in order to foster stable and sample-efficient learning in real-world applications where data collection is expensive or limited, such as robotics, autonomous driving, or more generally in the context of long-horizon planning.

Overview. HIRO [2] decomposes the main task Γ into a two-level hierarchy : one high-level policy π_Γ that sets subgoals g_ω every c timesteps, and a low-level policy π_ω that executes primitive actions to achieve those subgoals. Both policies are trained off-policy using experience replay. The key innovation lies in correcting the high-level goals during training to ensure consistency with the low-level behavior. This is achieved by re-labeling past goals using a learned inverse model and a heuristic that estimates what goal the low-level policy was actually pursuing. As a result, HIRO maintains the benefits of hierarchical abstraction while leveraging the efficiency of off-policy learning.

4.4.2 Option-Critic

Why It Matters. Options-Critic [3] is an implementation of policy-tree HRL it demonstrates how hierarchical structure can be discovered without manual design. It has shown promising results in continuous control, navigation, and exploration tasks. By automatically learning when to switch between

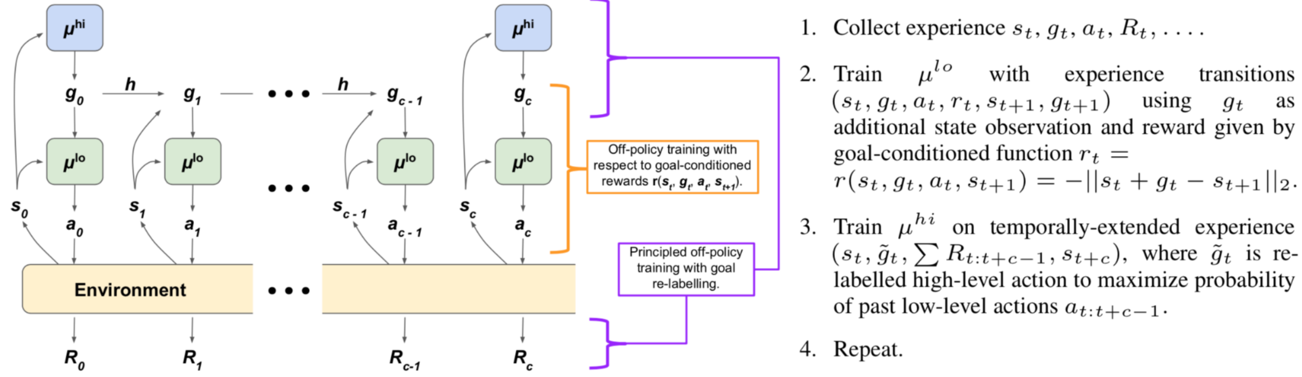


Figure 4.1: Illustration of the HIRO architecture. Figure from [2].

skills and which skills to refine. It provides a scalable pathway toward lifelong learning in robotics and has therefore been the base of many algorithm.

Overview. By integrating the option framework into a policy gradient setting, the **Options-Critic** architecture [3] enables the simultaneous learning of intra-option policies π_ω, their termination functions β_ω, and the top-level policy π_Γ. Unlike manually designed options, the Options-Critic allows an agent to automatically discover useful temporal abstractions during training.

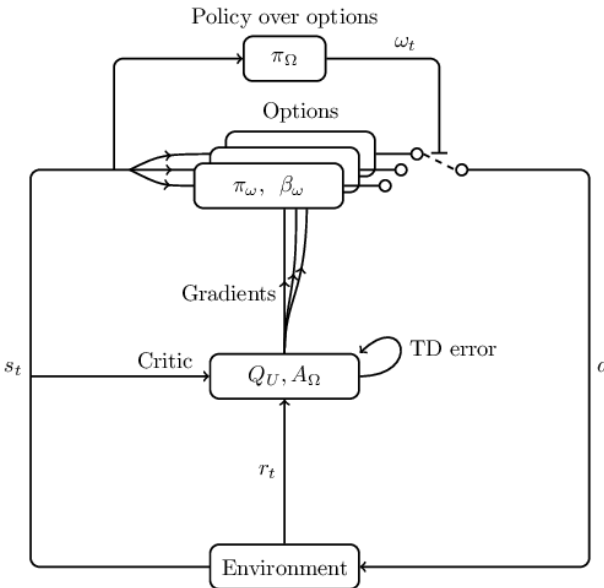


Figure 4.2: Illustration of the option-critic architecture. Figure from [3]

Chapter 5

Motivation

The limits of RL detailed in 3.1.2 has led to the development of HRL with two paradigm feudal and policy tree which allowed the option framework. The option framework obtains great result however one of his key challenge remains subtask discovery. This chapter aims to summarize the limits of the method detailed in 4 and the problem we aim to solve.

5.1 Limit of existing methods

Even though Hierarchical Reinforcement Learning (HRL) has made big strides, current methods for automatically finding subtasks still face major issues. These make it hard to use HRL in real-world robot control. The problems affect all kinds of approaches, whether they try to find subtasks in a unified way (UNI), separately (ISD), or by using LLM.

Unified Subtask Discovery (UNI). Current UNI methods aim to jointly discover subtasks and learn hierarchical policies from trajectory data. While attractive for their end-to-end design, these approaches often suffer from several drawbacks. First, they typically rely on strong assumptions about the task structure (for example predefined number of options or fixed horizon), which reduces their flexibility in unstructured settings such as locomotion over rough terrain. Second, UNI methods tend to produce opaque subtask boundaries: the discovered options are mathematically valid but difficult to interpret or reuse beyond the training task. Finally, learning both segmentation and sub-policies simultaneously is computationally demanding and often unstable, limiting their scalability to real-world robotic platforms.

Independent Subtask Discovery (ISD). Current ISD approaches attempt to identify reusable subtasks in a task-agnostic manner, usually during a pre-training phase. These methods whether graph-based bottleneck discovery, clustering, or latent skill embedding, contribute to modularity and transferability but encounter their own difficulties. They need a lot of training data that are not always available. Moreover, graph-based and clustering methods scale poorly in high-dimensional continuous spaces, while latent embedding techniques require vast exploration data and yield representations that are difficult to interpret. As a result, the discovered skills may be either too coarse to guide fine-grained locomotion behaviors or too abstract to integrate seamlessly into an HRL pipeline. Finally, ISD approaches often lack mechanism to ensure that subtasks are aligned the objectives of the downstream task, leading to inefficiencies when applied to embodied agents.

Foundation Model-based Approaches. First, **reward feedback methods** require extensive offline datasets or costly computation, making them difficult to scale to continuous-control domains like locomotion and nearly impossible in embedded system. Moreover, the correlation between such rewards and optimal value functions, while useful for credit assignment, does not automatically yield interpretable or reusable subtask structures. **Embedding similarity approaches** provide goal alignment rather than temporal decomposition. As such, they fall short in identifying intermediates milestones or subtasks, which are critical for long-horizon control tasks like stepping locomotion, where success hinges on structured sequences of movements. Finally, **reward-as-code paradigms** leverage symbolic representations to generate executable reward functions, thereby avoiding the need for parametric reward models.

However, their reliance on organized environment features limits their applicability to high-dimensional sensory domains (for example proprioceptive inputs in robotics). Although recent work attempts to infer symbolic abstractions from raw observations, these methods typically require many expert demonstrations and remain impractical for autonomous deployment in robotic locomotion. Nonetheless our work will take inspiration from these methods to apply it in our context while we will try to limit the need of expert demonstration thanks to the hierarchical decomposition.

5.2 Problem statement

This internship explores how Hierarchical Reinforcement Learning (HRL) can help robots learn to walk more efficiently and deal with harder tasks. The project, titled *Automatic subtask discovery and concurrent learning of subtasks for stepping locomotion using hierarchical reinforcement learning*, strive to teach robots advanced walking skills by breaking the process into smaller steps and learning them in a smart, organized way over time.

The central research question is:

> *How can we automatically discover meaningful subtasks from one demonstration trajectory and train corresponding sub-policies concurrently, in order to build an effective hierarchical agent for stepping locomotion?*

In order to answer this problematic, I will structure my work around three directions :

1. **Evaluating HRL advantages over standard RL** We assess the benefits of hierarchical architectures compared to flat reinforcement learning in terms of sample efficiency, scalability, and performance in long-horizon, sparse-reward locomotion tasks.

2. **Parallelized versus sequential subtask learning** This axis aims to look at the influence of training speed, resource use, and the quality of the learned behaviors of parallelized and sequential training.

3. **Trajectory segmentation and reward design using the option framework** Building on the knowledge gained from the two previous axes, we propose a weakly supervised RL pipeline. This pipeline aims to segment demonstration trajectories into coherent behavioral tasks. Each segments will define a subtask in the context of the Option framework, with the associated reward signals and end conditions, thus allowing the simultaneous training of low-level controllers.

Overall, this project aims to develop a scalable and interpretable HRL pipeline that supports automatic subtask discovery and concurrent sub-policy learning, tailored to the challenges of stepping locomotion for wheel-legged robots.

Chapter 6

Methodology

This chapter focuses first on presenting in details the environment that will be use across all experiments. Then it will detail the methodology of the comparison between RL and HRL and between sequential and parallelized training. Finally, it will present the development of the LLM pipeline for subtask discovery.

6.1 ARTER Environment

The ARTER environment is a physically simulated setup designed to evaluate locomotion and obstacle negotiation capabilities of a legged robot. It consists of three primary components:

- **The ARTER Robot** : a multi-jointed legged agent capable of continuous control over its joints. Its morphology and control architecture are detailed in Section 2.2.
- **The Ground** : a long, narrow parallelepiped from which the robot can fall off either side, introducing a natural constraint on balance and stability.
- **The Obstacle** : a smaller parallelepiped on the ground with variable height to challenge the robot's ability to step over it. The obstacle configuration is illustrated in Figure 6.1.

The different components were place in order to simulate a stepping locomotion task : the robot begin at one end of the platform which made up the ground and must reach a designated goal position located beyond the obstacle. Thus the robot has to step over the obstacle to reach the goal position.

Implementation The **Arter Env** is a simulation environment developed in **Python**, designed for single-agent interaction. At its core, it utilizes the **Genesis** library as its simulation engine, providing a robust and flexible framework for modeling agent behavior and environmental dynamics.

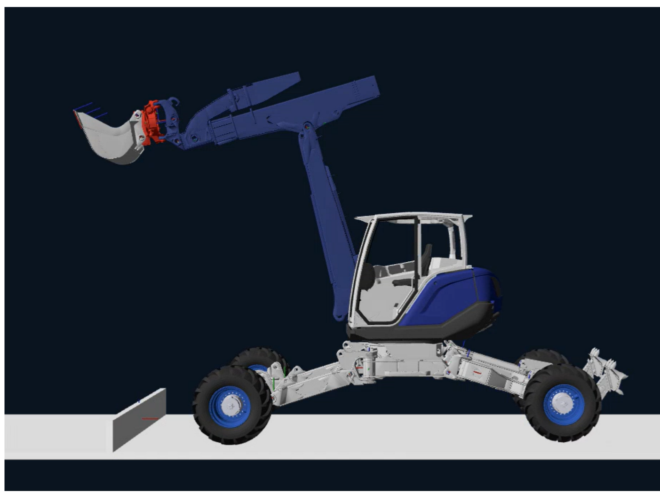


Figure 6.1: Image from the simulation of ARTER, the ground and the obstacle

The environment updates every **0.2 seconds**. It is built to scale well and includes **built-in support for running multiple environments at once**, which helps speed up training.

The observation space is a simple box of size $n_{\text{env}} \times n_{\text{observation}}$, where $n_{\text{observation}}$ corresponds to position (3), velocity (3), rotation position (4), and rotation speed (3); the position (23) and velocity (23) of each joint; the position of the shovel with respect to the ground (3); and the distance between the robot and the obstacle (3), for a total of 64 observations. A detailed breakdown of the observation space can be found in Table A.3.

The action space has been implemented as a box of size $n_{\text{env}} \times n_{\text{action}}$, corresponding to the velocity (23) of each joint. A detailed breakdown of the action space can be found in Table A.2.

To make the environment as modular as possible, various wrappers have been implemented. A summary of these wrappers is provided in Table A.4.

Simulation platform selection Preliminary experiments were conducted using **PyBullet**, owing to its lightweight implementation and ease of use. However, training reinforcement learning policies in PyBullet proved prohibitively slow.

In order to tackle the limitation in term of performance, we considered the use of GPU-based simulators, that can support parallelization. In particular, the following three options were reviewed:

- **MuJoCo**: Known for high-fidelity physics modeling and low-latency simulations.
- **Isaac Gym**: Developed by NVIDIA, providing GPU-parallelized simulation with support for thousands of concurrent environments.
- **GENESIS**: A newer platform focused on scalability and fine-grained control of hierarchical and structured tasks.

At the end, we choose to select genesis because of four differents reasons :

1. **Parallel Simulation Efficiency:** GENESIS concurrent environments on a single GPU significantly outperforming PyBullet and MuJoCo in benchmarks.
2. **Modularity:** The platform enables custom reward functions, and modular agent design, which are essential for our hierarchical reinforcement learning framework.
3. **Low-Level Control Access:** GENESIS offers granular access to simulation internals such as joint states, contact dynamics, and terrain conditions.
4. **Acceptable Trade-Offs:** Although GENESIS lacks some high-level abstractions and user interfaces provided by more mature platforms, the overhead was manageable. Certain components (for example manual physics parameter tuning, logging) required bespoke implementation, but the trade-off was justified by the gain in simulation speed and flexibility.

Implementation strategy to structure observation and action space In order to parallelize effectively the environment and be able to adapt to the constraint multiple strategy to structure the action and observation space has been developed :

- Pure `TensorDict` Usage : The first approach involved leveraging the standard `TensorDict` API provided by TorchRL to structure observations, actions, rewards, and auxiliary data. This method ensures maximum compatibility with built-in components such as environments, loss functions, and collectors.
- Tensor with sliced views in `TensorDict` : In the second approach, a single contiguous tensor was used to store all data, and slices from this tensor were assigned as views within a `TensorDict`. This design aimed to retain TorchRL compatibility while enabling fast scalar operations.
- Custom strategy: Tensor with Index-Based Feature Mapping : In the second approach, a single contiguous tensor was used to store all data, and slices from this tensor were assigned as views within a `TensorDict`. This design aimed to retain TorchRL compatibility while enabling fast scalar operations.

- Wrapper class combining tensor, index map and optional `TensorDict` : The last method, that were chosen in the project use the previous Custom Strategy and also introduces a lightweight wrapper that encapsulates the raw tensor, index mapping logic, and optional conversion to `TensorDict` in order reconcile low-level performance with high-level framework integration.

The table 6.1 allow to understand the advantnages and limitation of each method.

Aspect	Pure `TensorDict`	Tensor + Sliced Views	Tensor + Index Map	Tensor + Index Map + Wrapper Class
TorchRL Compatibility	High	Moderate	Low	High (via wrapper)
Performance	Low for bulk ops	High for scalar ops	Very high	High
Memory Efficiency	Moderate	High	High	High
Structural Flexibility	Moderate	Low (layout constrained)	Very high	High
Ease of Feature Access	Moderate (recursive)	Moderate (view-sensitive)	High (index-based)	High
Safety	High	Moderate	Low	Moderate to High
Ease of Maintenance	High	Moderate	Low	Moderate

Table 6.1: Comparison of data structuring strategies in TorchRL pipeline

6.1.1 Action

Overview Each action represents a set of continuous velocity commands sent to the robot's joints. These actions are continuous and bounded within safe operating limits.

Formalization The action space $\mathcal{A}$ consists of continuous control signals applied to each joint of the robot. Specifically, the agent outputs desired joint velocities for all controllable degrees of freedom.

Let:

- n_{joints}: number of actuated joints
- $a \in \mathbb{R}^{n_{\text{joints}}}$: action vector representing desired joint velocities

Then the action space is defined as:

$$\mathcal{A} = \{a \in \mathbb{R}^{n_{\text{joints}}} \mid a_j \in [a_{\text{min}}, a_{\text{max}}] \quad \forall j = 1, \ldots, n_{\text{joints}}\}$$

Where:

- a_j: desired velocity for joint j
- $[a_{\text{min}}, a_{\text{max}}]$: allowable velocity range for each joint, typically symmetric around zero

The agent's policy $\pi(a \mid s)$ maps observations $s \in \mathcal{S}$ to actions $a \in \mathcal{A}$, aiming to maximize cumulative reward while satisfying physical and behavioral constraints.

Implementation The action space is dynamically constructed based on the robot's configuration and the degrees of freedom (DOFs) selected for control with only subset of joints that can be actuated via position or velocity commands. The environment aggregates the bounds during the initialization for each controllable DOF, forming the global action space. These bounds are used to normalize the agent's output to the range $[-1, 1]$, which are then scaled back to the physical limits defined in the robot's configuration. In order to allow precise control over individual DOFs, each action component is indexed and mapped to its corresponding joint.

At each step, the agent's action vector is interpreted and dispatched to the robot. If position control is enabled, the robot try to reach the given position. To match the joint's velocity limits, the agent's output is also scaled before being applied. It should also be noted that the system was adapted to support frozen or externally controlled joint.

In order to prevent instability or physical violations, the environment enforces joint limits after each applied action by clamping joint positions and velocities within their previously configured bounds. This design has been chosen to allows fine-grained and configurable control over which joints are actuated, frozen, or externally driven, making the framework adaptable to a wide range of robot morphologies and control strategies.

Specification for Option-Critic As Option-Critic algorithm that we used only works with discrete action, we have to discretize the action space. Therefore we create a dedicated wrapper that for each continuous action, allows the agent to choose between three discrete actions -1, 0 or 1 with a particularity for the wheels. Indeed as the robot might need to activate all wheels at the same time, three extra discrete actions has been added that allow to control the four wheels at the same time.

6.1.2 Observation

Overview At each timestep, a structured observation vector made of information about the robot itself and its surroundings is generated. Specifically, the robot perceives its base position and velocity, orientation represented as a quaternion and angular velocity. The vector also includes the position and velocity of each joint. Additionally, the robot measures distances to the ground and to the obstacle in the environment. In order to harmonize scaling across tasks and environments, the different element of the observation vector are normalized to the range $[-1, 1]$ using predefined configuration parameters. This normalized feedback allows the agent to interpret its state more efficiently. The detail of the observation space is given in A.3

Formalization The observation space $\mathcal{S} \subseteq [0, 1]^d$ consists of normalized sensory data available to the agent at each timestep. Let:

- $s \in [0, 1]^d$: normalized observation vector of dimension d
- $s = [s_1, s_2, \ldots, s_d]$: individual components of the observation

Each raw signal x_i is normalized as:

$$s_i = \frac{x_i - x_{\min}^{(i)}}{x_{\max}^{(i)} - x_{\min}^{(i)}}, \quad \text{where } x_{\min}^{(i)}, x_{\max}^{(i)} \text{ are bounds from the config}$$

In order to enable the agent and interpret its state consistently across different scenarios and select appropriate actions accordingly, the observation space is normalized

Implementation The observation space is constructed based on the robot configuration and the selected observation modalities. Most of the required signals, such as base position, orientation, velocities, and joint states, are directly accessible from the Genesis simulation. At the start, the observation space is set up by collecting the minimum and maximum values for each signal, based on the robot and its joints (DOFs). These values are used to scale the signals into the $[0, 1]$ range. Each signal is given a unique spot in the observation vector, so it can be easily read and updated during simulation.

In addition the agent receives feedback in the form of distances to key environmental entities, such as the ground and the unique obstacle. These distances are computed using a dedicated wrapper class that supports axis-specific and norm-based measurements. Each distance is offset and normalized according to user-defined parameters, and then appended to the observation vector. The wrapper ensures that the distance signals are smoothly combined into the overall observation space, with proper indexing and normalization. This design has been chosen to allow the agent to perceive spatial relationships in a compact and scalable format.

6.1.3 Reward

Overview In order to help the legged robot evolute more effectively and safely in the environment, we have created a custom reward system that combines three behavior rules with a main reward :

- **Stability:** Promotes an upright posture and penalizes deviations from a balanced orientation. Ensures the robot avoids excessive tilting or falling.
- **Energy Efficiency:** Discourages unnecessary joint effort by penalizing torque-velocity product in order to save energy.
- **Contact Smoothness:** Rewards stable ground interaction by discouraging abrupt vertical center-of-mass movements. Helps reduce bouncing and erratic contact forces.
- **Goal Proximity (Main Reward):** Provides the primary task objective by rewarding closeness to a predefined target location.

Adaptive Weighting: Instead of fixing reward weights, we dynamically compute coefficients based on the mean value of each constraint reward. Intuitively, underperforming constraints are upweighted while strong ones are downweighted. Coefficients are normalized and bounded below by a minimum threshold to ensure fairness among constraints.

Formalization The total reward function consists of a goal-driven term and a weighted sum of auxiliary constraints:

$$R(s,a) = 10 \times C_{\text{goal}}(s) + \sum_{i=1}^{3} \lambda_i \cdot C_i(s,a)$$

where the coefficients λ_i are computed adaptively at each training step from normalized inverse mean rewards:

$$\lambda_i = \frac{\frac{1}{\mathbb{E}[C_i]} + \delta}{\sum_j \left(\frac{1}{\mathbb{E}[C_j]} + \delta\right)}, \qquad \lambda_i \geq \lambda_{\min}$$

with δ a small constant to avoid division by zero.

Stability Constraint Aims to encourage a stable locomotion. Reward is maximal when upright, decreases smoothly with tilt angle (ignoring yaw rotation):

$$C_1(s) = \max\left(0,\ 1 - \frac{\theta(s)}{\pi/2}\right)$$

where $\theta(s)$ is the tilt angle from the upright quaternion.

Energy Constraint Aims to encourage efficient control by penalizing torque–velocity effort:

$$C_2(s,a) = \frac{1}{1 + \alpha \sum_{j=1}^{n_{\text{joints}}} |a_j \cdot \dot{q}_j(s)|}$$

with scaling factor $\alpha > 0$.

Contact Constraint Aims to discourage abrupt vertical center-of-mass motion:

$$C_3(s) = \frac{1}{1 + \beta\,|v_z(s)|}$$

where $v_z(s)$ is the vertical COM velocity, and $\beta > 0$ is a scaling parameter.

Goal Constraint (Main Reward) Aims to provides the primary task objective, rewarding closeness to the target:

$$C_{\text{goal}}(s) = \max\left(0,\ 1 - \frac{\|p_{\text{com}}(s) - \mathbf{g}\|}{d_{\max}}\right)$$

where $p_{\text{com}}(s)$ is the COM position, $\mathbf{g}$ the target position, and $d_{\max}$ the maximum distance threshold.

Implementation The reward function is implemented as a modular wrapper around the environment. Dedicated classes that define an `evaluate()` method encapsulated each constraints by computing their reward contribution based on the current observation and action.

During each environment step, the wrapper computes the goal reward independently and evaluates all constraint rewards in parallel. The constraint values are collected into a reward matrix, and weights are calculated by taking the inverse of their average values. This way, constraints that are not doing well get more attention, while those already working fine are given less weight. A minimum weight is set to make sure no constraint is completely left out. The final reward is calculated by adding the weighted constraint rewards to the goal-related reward.

In order to facilitates reward shaping and debugging, detailed reward diagnostics like individual constraint values, their adaptive coefficients, and the total reward breakdown are logs into the environment's `info` dictionary thanks to a wrapper.

6.1.4 Termination

Overview When the robot reaches a goal or violates a critical constraint the episode ends automatically in order to avoids prolonged periods of ineffective behavior. The conditions that can trigger a termination are the robot loses balance and falls off the walking surface, a collision occurs with the environment or the robot itself or the robot successfully reaches the goal position beyond the obstacle.

Formalization An episode is said to terminate when a predefined condition is met, indicating either task completion or failure. Let $s_t \in \mathcal{S}$ be the observation at timestep t. The termination condition is defined by a boolean function:

$$T(s_t) = \begin{cases} 1 & \text{if } s_t \in \mathcal{S}_{\text{terminal}} \\ 0 & \text{otherwise} \end{cases}$$

Where $\mathcal{S}_{\text{terminal}} \subset \mathcal{S}$ is the set of terminal states.

Implementation In the case of the ARTER robot in the class the reset method return for a given env if one the following condition is met : the number of env exceed the maximum set in the configuration file of the env, the collision wrapper detect a collision between the robot and the obstacle or the robot fall from the platform.

6.2 Comparison of RL and HRL algorithm

The aim of this study was to assess the practical advantages and drawbacks of Hierarchical Reinforcement Learning (HRL) compared to standard RL approaches. To this end, we evaluated three representative algorithms: PPO for RL, HIRO for feudal HRL (see 4.4.1), Option–Critic for Policy-Tree (see 4.4.2). This comparison served two purposes: (i) to establish a robust performance baseline across approaches, and (ii) to motivate the making of a better, segmentation-based subtask discovery pipeline.

The experiments were conducted on the ARTER environment (Section 6.1). All algorithms were trained under identical computational budgets and environment configurations. Hyperparameters were standardized across methods wherever possible to assure an impartial and unbiased comparison.

This part will focus on evaluate the three following criteria :

1. **Learning performance** : The learning performance is evaluated thanks to the reward value at the end of the training.
2. **Sample efficiency** : The sample efficiency assessed through reward progression over time.
3. **Interpretability and subtask reuse** : The interpretability will be analysed through litterature review and qualitative inspection of learned behaviors and the reusability of discovered subtasks.

The insights from this analysis were then used to guide the development of our proposed segmentation-based subtask discovery pipeline, presented in Section 6.4.

Implementation To run the experiments, we used open-source versions of PPO [32], HIRO [33], and Options-Critic [34] from their GitHub pages. We adjusted each algorithm so it could work with the ARTER environment (see Section 6.1), making sure they all used the same observation and action spaces.

We developed a unified Python framework comprising custom `Trainer` and `Tester` classes to standardize training and evaluation procedures. All algorithms were trained under identical computational budgets, with hyperparameters aligned wherever possible to minimize confounding factors. Each experiment aims to be repeated 10 times with different random seeds to account for stochasticity in training.

This setup made it possible to compare how the algorithms behave in a reliable and repeatable way. It also gave practical support for the segmentation-based subtask discovery pipeline.

6.3 Comparison of sequential and parallelized training

These part aims to verify the efficiency of parallelized environment execution compared to sequential one. We also assess the scalability of parallelization. Therefore two configurations were evaluated:

1. **Sequential execution** : a single environment instance running step-by-step interaction with the agent.
2. **Parallelized execution** : multiple environment instances running concurrently

The goal of this comparison is to quantify the trade-off between throughput and learning efficiency. Parallelized environments can drastically reduce wall-clock time and increase the volume of collected experience per unit of time. However, they may also introduce challenges, such as reduced temporal coherence between sampled trajectories, which could impact training.

The comparison focuses on two primary aspects:

- **Scalability in wall-clock time** : measuring how the number of parallel environments affects simulation throughput.
- **Impact on learning performance** : comparing maximum achieved reward using PPO.

This study provides a secondary performance baseline, complementing the RL vs. HRL comparison, and informs the decision to integrate parallelized execution within the proposed pipeline.

Implementation To empirically evaluate the effects of execution mode, we implemented both sequential and parallelized training pipelines using PPO as representative algorithms (as it was supposed to be use in the option-framework of our pipeline). The sequential and parallelized setup utilized instances of the ARTER environment, ensuring strict temporal consistency.

Performance metrics included the speed of the training based on the number of episode in one hour and the learning efficieny based on the max reward reach.

This analysis complements the RL vs. HRL comparison and supports the integration of parallelized execution in our segmentation-based pipeline in order to improve efficiency.

6.4 LLM Pipeline

The goal of the proposed pipeline is to make hierarchical reinforcement learning (HRL) easier by combining human or AI-generated demonstrations with automatic subtask discovery. These demonstrations are split into clear sub-tasks, labeled, and given reward functions using large language models (LLMs). This helps reduce the need for expert input and makes the learned behaviors easier to understand and reuse.

The pipeline was originally planned to be tested in the ARTER environment qualitatively and quantitatively. The quantitative part would check how well and consistently the sub-tasks perform during training, while the qualitative part would look at how clear and useful the segmentations and rewards are.

However during our test, the automatically generated rewards didn't lead to successful training. Thus we unfortunately couldn't make a full numerical comparison. Instead, we focused on a qualitative analysis, looking at how meaningful the segmentations were, how clear the sub-task labels looked, and whether LLM-generated rewards could fit into HRL workflows. These insights still help highlight the limits of the current method and suggest ways to improve it in future work.

The pipeline that we developed is compose of five modules which operate sequentially (the order of presentation is the order of call of each module in the pipeline) :

- **Trajectory Recorder** : collects raw demonstrations from human operators or pre-trained agents.
- **Trajectory Segmenter** : partitions trajectories into coherent sub-tasks using algorithmic or heuristic segmentation methods.
- **Segment Annotator** : uses LLMs to generate semantic descriptions for each segment, providing interpretable labels.
- **Reward Creation** : derives subtask-specific reward functions based on segment annotations and environment feedback.
- **Hierarchical Training** : trains an HRL agent using the Options framework, orchestrating learned sub-tasks into a complete policy.

This modular design aims to allow to replace easily each component. Indeed as we separate demonstration processing from hierarchical training, the pipeline can be adapted to a variety of environments and HRL algorithm.

Implementation We implement the pipeline in Python using a Conda-managed environment in order to ensure reproducibility. Moreover Conda allows version control, and consistent dependency resolution across systems. Our goal was to make a modular architecture. Indeed each stage segmentation, annotation, reward inference are encapsulated in its own dedicated class. All the different components are orchestrated by a central pipeline manager. The pipeline manager coordinates execution flow and ensures a smooth integration between modules.

The development of our pipeline also aims to minimize manual intervention between stages. To achieve this, inter-module communication is handled through a structured file system, anchored by a master JSON configuration file. The JSON file stores all relevant metadata, including robot specifications, environment parameters, segment boundaries, annotation prompts, and reward outputs. Each module reads from and writes to this shared configuration, allowing the pipeline to operate end-to-end without requiring the user to manually copy, paste, or reformat intermediate results.

For example, once the trajectory segmentation module generates a nested JSON hierarchy, the annotation module automatically extracts the relevant timestamps and video clips, applies the semantic labeling model, and appends the results back into the same JSON structure. Similarly, the reward inference module accesses the annotated segments and robot description directly from the configuration file, generates reward specifications using a large language model, and stores them in a format immediately consumable by the training module.

The automation of the pipeline required however to be careful with data formatting to avoid miscommunication between the module and has required a lot of debugging. But the efficiency gain in terms

of experimentation and iteration as running the pipeline requires only minimal configuration were worth the effort.

6.4.1 Save trajectory

Overview The first of the pipeline consist of a record of a demonstration of the target behavior, in the context of the internship the stepping locomotion in the ARTER environment. The record can come from a human or an existing algorithm and the trajectory can include both a video and detailed logs of states, actions, and rewards. This data forms the base for the next steps of the pipeline like segmentation and learning, indeed starting with meaningful examples helps the system learn faster and generalize better, without needing to explore everything from scratch.

Formalization A demonstration is collected from the environment $\mathcal{E}$, either by a human or an external agent. The recorded trajectory consists of two components:

$$\tau = (\mathcal{V}, \mathcal{O}) \quad \text{where} \quad \mathcal{V} = \{v_t\}_{t=0}^{T}, \quad \mathcal{O} = \{s_t\}_{t=0}^{T}$$

Here, $\mathcal{V}$ is a sequence of video frames, and $\mathcal{O}$ is a log of observed states $s_t \in \mathcal{S}$ at each time step. No actions or rewards are recorded at this stage. This observation-only trajectory serves as the raw input for segmentation and semantic interpretation.

Implementation The recorder was implemented as a custom Gym wrapper in python designed to capture trajectories during environment interaction. Recorded data is stored in CSV format to ensure interoperability and facilitate manual inspection. Internally, all data is temporarily buffered in PyTorch tensors for efficient batch processing, which are flushed to disk at intervals specified in the configuration. A dedicated configuration class, defined in JSON format, allows flexible control over the logging behavior. This includes options to append metadata such as log identifiers (row indices), timestamps (relative to simulation time), and selection of specific observations and actions to record.

6.4.2 Segmentation

Overview The **Segmentation Module** processes the raw demonstration to identify temporally coherent segments. Each segment is supposed to represent different subtask that can be learned independently. The segmentation process produces a hierarchy of segment, structured as a tree-graph. This hierarchy has to reflect the logical organization of the task. This module is key to isolate interpretable units of behavior, which can later be reused or transferred across different tasks and environments.

This module segments the log acquired by the previous module by analyzing temporal patterns or discontinuities in the state-action trajectory τ.

For the evaluation, we compare the output to a ground truth. As the segmentation algorithms typically output a list of change points (timestamps or indices). We will treat each predicted change point as a positive prediction. A prediction is considered correct (true positive) if it falls within a small window around a ground truth point. Points outside any window are false positives. Ground truth points not matched by any prediction are false negatives.

Formalization The trajectory $(\mathcal{V}, \mathcal{O})$ is decomposed into a hierarchy of segments across multiple levels. Let L be the maximum depth of the hierarchy, defined by the algorithm. At level $\ell = 0$, we have the root segment:

$$\tau^{(0)} = (\mathcal{V}^{(0)}, \mathcal{O}^{(0)}) = (\mathcal{V}, \mathcal{O})$$

Each segment $\tau_i^{(\ell)}$ at level ℓ is recursively partitioned into $K_i^{(\ell)}$ child segments at level $\ell + 1$:

$$\tau_i^{(\ell)} = \bigcup_{j=1}^{K_i^{(\ell)}} \tau_{i,j}^{(\ell+1)}$$

where each $\tau_{i,j}^{(\ell+1)} = (\mathcal{V}_{i,j}^{(\ell+1)}, \mathcal{O}_{i,j}^{(\ell+1)})$ is a temporally bounded subsegment of its parent. The process continues until $\ell = L$, at which point the segments are considered atomic and ready for annotation and learning.

The full hierarchy can be represented as a tree $\mathcal{H} = (\mathcal{N}, \mathcal{E})$, where:

- $\mathcal{N}$ is the set of all segments $\tau_i^{(\ell)}$ across levels $\ell = 0, \ldots, L$
- $\mathcal{E}$ contains edges $(\tau_i^{(\ell)}, \tau_{i,j}^{(\ell+1)})$ representing parent-child relationships

This recursive structure enables multi-scale decomposition of complex tasks into interpretable and modular subcomponents.

Implementation First the raw trajectory data (in CSV format), are preprocessed by the segmentation module. This preprocessing uses `pandas` and `numpy` for efficient data manipulation.

To construct a hierarchical segmentation, we recursively apply PELT to each identified segment, starting from the full trajectory as the root.

PELT (Pruned Exact Linear Time) is an algorithm that splits a long sequence of data into meaningful segment. In our context, the data are log from the ARTER movement record acquired by the previous module. The goal of PELT is to minimize a penalty in order to find the best segmentation keeping each segment coherent and avoiding too many cut. As PELT has a linear complexity it is ideal on our context for analyzing long demonstrations offline, helping identify clear phases of behavior without manual effort.

This recursive decomposition aims to give a tree-like structure, where each node corresponds to a temporally bounded subsegment. The resulting hierarchy is serialized into a nested JSON format in order to save for each segment its start and end timesteps. However PELT proves inefficient to create a hierarchical segmentation (as it gives the same segment when apply to a segment of the initial log data), therefore we only apply PELT one time.

6.4.3 Segmentation annotation

Overview Each segment is passed to a **Semantic Annotation Module** designed to provide a natural language description for each subtask using both the video and log data, leveraging the LLM capacity to generalize across domains. To improve the relevance and accuracy of the annotations, contextual information, such as a description of the robot, its joints, or the task domain, can be included in the prompt.

Formalization Each atomic segment $\tau_i^{(L)} = (\mathcal{V}_i^{(L)}, \mathcal{O}_i^{(L)})$ is passed to a large language model $\mathcal{L}$ for semantic annotation. The model can uses both the video frames and observation log, along with contextual information $\mathcal{C}$ (for example robot description, task domain), to generate a symbolic or natural language label:

$$\alpha_i^{(L)} = \mathcal{L}(\mathcal{V}_i^{(L)}, \mathcal{O}_i^{(L)}, \mathcal{C})$$

These annotations describe the sub-tasks represented by the leaf segments and serve as a basis for reward generation.

Implementation To semantically annotate each segment, we developed a Python class that ingests both the raw video and the nested JSON hierarchy produced by the segmentation module. Using the segment timestamps, the class extracts corresponding video clips for each subtask using the `MoviePy` library. These clips are then passed to the `VideoRefer` model `DAMO-NLP-SG/VideoLLaMA3-7B`, a large multimodal language model capable of generating natural language descriptions from video input.

Each segment is annotated using a structured prompt that includes contextual information about the robot (in the form of its URDF file) and task domain, to enhance the specificity and relevance of the generated descriptions. The prompt is formulated as follows:

```
You are a specialist in robotics. Here is a description of the robot: [robot
description]. Analyze as precisely as possible what the robot is doing in this
video. Focus on its goal, joint movements, and end-effector behavior.
```

"[robot description]" is a paramater of the class, in the case of ARTER, the robot description corresponds to a minimal URDF file of the robot.

The model's output is then embedded into the corresponding node of the nested JSON structure, enriching each segment with a high-level semantic label.

6.4.4 Task reward

Overview Following annotation, a **Reward Inference Module** assigns scalar rewards to each segments and subsegment, facilitating HRL-compatible subgoal definitions. This model uses the annotated descriptions and, optionally, the raw trajectory data to infer what constitutes successful behavior within the segment. The output is a python code-based reward. Automating reward design addresses one of the key bottlenecks in reinforcement learning, reducing the need for handcrafted reward engineering and ensuring that the learning signal is aligned with the semantic goals of each sub-task.

Formalization Using the annotation $\alpha_i^{(L)}$, a reward function $R_i^{(L)}$ is synthesized by a second language model $\mathcal{L}_R$:

$$R_i^{(L)} = \mathcal{L}_R(\alpha_i^{(L)}, \mathcal{O}_i^{(L)})$$

The reward function $R_i^{(L)} : \mathcal{S} \to \mathbb{R}$ maps observed states to scalar rewards. Since actions are not available, the reward must be inferred from state transitions and semantic goals. This step enables reinforcement learning without manual reward

Implementation To automate reward generation, we developed a Python module that ingests a nested JSON file containing semantic annotations for each segment and subsegment of the trajectory and use LLaMa 3.3.

The reward inference process follows a clear prompt that contains key instructions and information :

- A minimal URDF-based description of the robot (ARter).
- A summary of the robot's environment. In the summary we explain notably the controllable joint velocities and observable joint positions. A development is then made of the different entities of the environment such as the robot itself, the ground, and obstacles.
- A description of the intended behavior of the sub-task obtained though the semantic annotation module.

The prompt follows this template:

```
You are a robotics expert. The robot is described as follows: [robot description].
In the robot's environment, we can control the speed of any joint and observe the
velocity and position of any joint, as well as entities like the robot, ground,
or obstacles.
Could you generate a very simple reward function for a reinforcement learning algorithm
to train the robot to perform the following task: [annotation]?
```

The model's output is a scalar reward specification tailored to the annotated behavior written like a Python expressions.

6.4.5 Training

Overview In the final stage of the pipeline, each segment is trained independently using the generated reward functions. The training is conducted within the framework of HRL, specifically using the Options paradigm. Each segment corresponds to an option, a temporally extended action with its own policy and termination condition. Once trained, these options are orchestrated by a high-level policy that selects and sequences them to accomplish the overall task. This structure enables temporal abstraction, improves sample efficiency, and enhances interpretability. Moreover, the modular nature of options is supposed to facilitate transfer to new tasks and environments.

Formalization Each leaf segment $\tau_i^{(L)}$ is associated with an option $\mathcal{O}_i^{(L)} = (\pi_i^{(L)}, \beta_i^{(L)})$, where $\pi_i^{(L)}$ is the intra-option policy and $\beta_i^{(L)}$ is the termination condition. The high-level policy π_H selects among options based on the current state:

$$\pi_H(o \mid s) = \text{probability of selecting option } o \text{ in state } s$$

Each sub-policy $\pi_i^{(L)}$ is trained to maximize the expected cumulative reward under $R_i^{(L)}$:

$$J(\pi_i^{(L)}) = \mathbb{E}_{s_0 \sim \mathcal{O}_i^{(L)}} \left[\sum_{t=0}^{T_i} \gamma^t R_i^{(L)}(s_t) \right]$$

The orchestrator π_H coordinates the execution of options to complete the overall task, enabling modular and interpretable learning across hierarchical levels.

Implementation To evaluate the effectiveness of the automatically generated reward functions, we aims to implement a two-stage reinforcement learning pipeline based on Proximal Policy Optimization (PPO). The first stage focuses on learning low-level behaviors for each leaf segment (meaning the more nested segment in the hierarchy) independently, while the second stage attempts to train a high-level controller to orchestrate these learned behaviors hierarchically.

In practice, training PPO on individual segments didn't work well because the automatically generated rewards weren't expressive or reliable enough. The behaviors that were learned were unstable and didn't carry over to new episodes. Because of this, we decided not to continue developing the hierarchical controller further.

This implementation was kept deliberately simple and does not use the full power of the Options framework, it only includes two levels of decision-making. The main reason for this was to quickly test and refine different ways of designing rewards, so the reward inference module could be improved step by step. In order to better leverage the modularity and temporal abstraction offered by the Options paradigm, future work should explore more sophisticated hierarchical architectures and training strategies

Chapter 7

Experimentation and evaluation

7.1 RL versus HRL

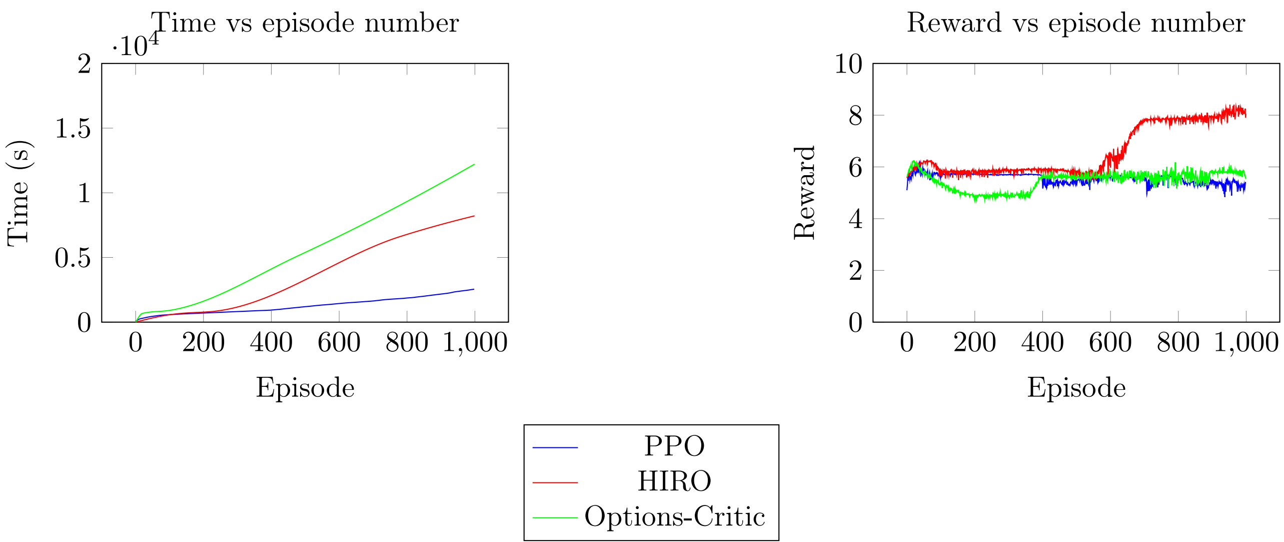


Figure 7.1: Comparison of algorithms across environments: time and reward

Performance analysis The results show that the tested algorithms behave differently and have distinct strengths and weaknesses. **Proximal Policy Optimization (PPO)** is the fastest to train. However PPO obtains the lowest performance, with a reward at the end of the training of 5.4. **Option Critic** appears deceptively promising, as it is approximately five times slower that PPO and only reaches a slightly higher reward of 5.7. Thus it is clear that the best algorithm seems **HIRO**, which achieves a reward of approximately 8, with only being approximately three times slower than PPO indicating a more effective learning process is **HIRO**.

The reward metric is based on the agent's distance to the goal, which lies beyond an obstacle. We can notice that HIRO was the only algorithm that succeed to reach the obstacle, though it did not surpass it, while the others struggled to make comparable progress within the same number of episodes. However we need to acknowledged that the result may reflect randomness rather than a robust trend, due to the limited number of training runs. Therefore in order to validate HIRO's apparent advantage over the PPO and Option-Critic repeated trials would be necessary guarantying statistical reliability.

Constraint satisfaction analysis We check how well each algorithm follow the three constraints : keeping balance of the robot, improving energy efficiently, and avoid contact with the obstacle. The hight the corresponding coefficient of the constraint the poorer the constraint is respected as the algorithm compensates for low reward by increasing that constraint's weight. If all constraint rewards converge around 0.33, it suggests that the constraints are being satisfied equally, but the constrains may not be well respected.

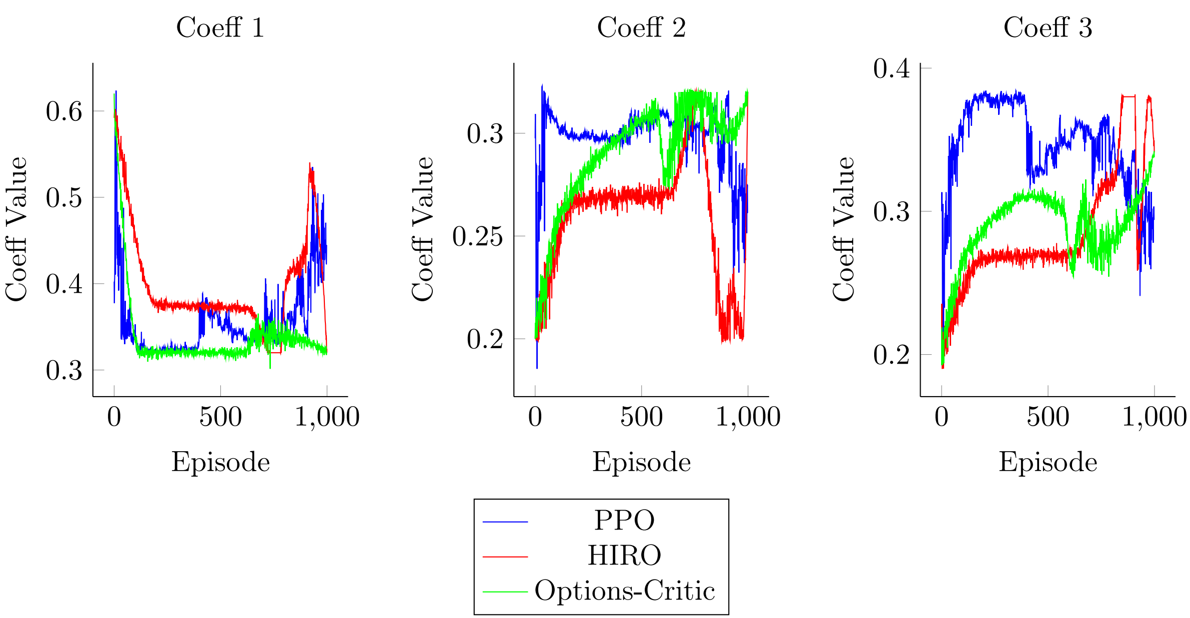


Figure 7.2: Comparison of four RL algorithms. Each subplot shows the evolution of the coefficient of each reward

At the beginning of the training, all algorithms struggle with the stability constraint. However they quickly adapt showing difficulty in following the energy and contact constraints. But as the training progresses, the constraint rewards gradually balance by converging to an equilibrium, indicating that the algorithms learn to respect the constraints more uniformly. Qualitatively all the constraints seems to be respected at the end of the training with the robot seems to be more balance and limiting useless movement therefore respecting the balance and energy constraint.

Interpretability A qualitative analysis has been made in order to assess the interpretability of each algorithm. The analysis is based on the literature from [2, 3, 6]. For PPO, as it is essentially a black box, there is no goal definition, but it is possible to interpret the output over the long run if you can determine what the robot is "trying to do". For HIRO, the interpretation was based on the subgoal of the high level; as the subgoal corresponds to the robot's position, it is possible to see where the lower policy has to go. For Option, one way to interpret the result is to see what each option is trying to do. However, in my context, the options didn't show interpretable behavior. This may be due to the fact that the option was quickly changing and that it is very difficult to determine the state in which each option should be active or not.

Algorithm	**Goal Interpretability**	**Output Interpretability**	**Transferability of Learning**
PPO (Proximal Policy Optimization)	Low	Medium	Low
Options-Critic	Medium	Medium	Medium
HIRO (Hierarchical Reinforcement Learning with Off-Policy Correction)	Medium	Medium	High

Table 7.1: Qualitative comparison of interpretability and transferability across RL algorithms

Conclusion In conclusion, our analysis revealed that despite the theoretical promise of HRL, some current approaches, such as option-critic, struggle in environments that are both complex and high-dimensional. Although our qualitative analysis shows HIRO tends to generate interpretable and trans-

ferable subtasks, its two-level hierarchy limits scalability when applied to long-horizon and complex environments. These insights directly guided the development of our segmentation-based subtask discovery pipeline, designed to overcome these specific limitations.

7.2 Sequential versus parallelized

Bottleneck analysis We aimed to evaluate the main bottleneck during training in the ARTER environment. Therefore, we ran a test on the T1550 GPU: we first tried to reach 1,000,000 steps using the ARTER environment implemented in PyBullet and Genesis. Then, we tested the PPO, HIRO, and Option-Critic architectures in order to fairly evaluate the algorithms without the environment slowing them down. To do so, we created a custom environment that returns a random observation of size 10 at each step.

Environment	**Time (s)**
ARTER Environment (Pybullet)	34000
ARTER Environment (Genesis)	16300
PPO	4600
HIRO	8400
Options-critics	11200

Table 7.2: Evaluation of Algorithm Speeds: Time to Reach 1,000,000 Steps (in Seconds)

We can analyze that the main bottleneck does not come from the algorithm itself, but from the environment. Therefore, improving the speed of each step for a given environment—such as using Genesis parallelization is a valuable strategy for enhancing efficiency.

Influence of Parallelization on the ARTER Environment The table 7.3 aims to evaluate the influence of parallelization on the overall performance of the ARTER environment. The test has been run on a GPU NVIDIA T1550 on the ARTER environement.

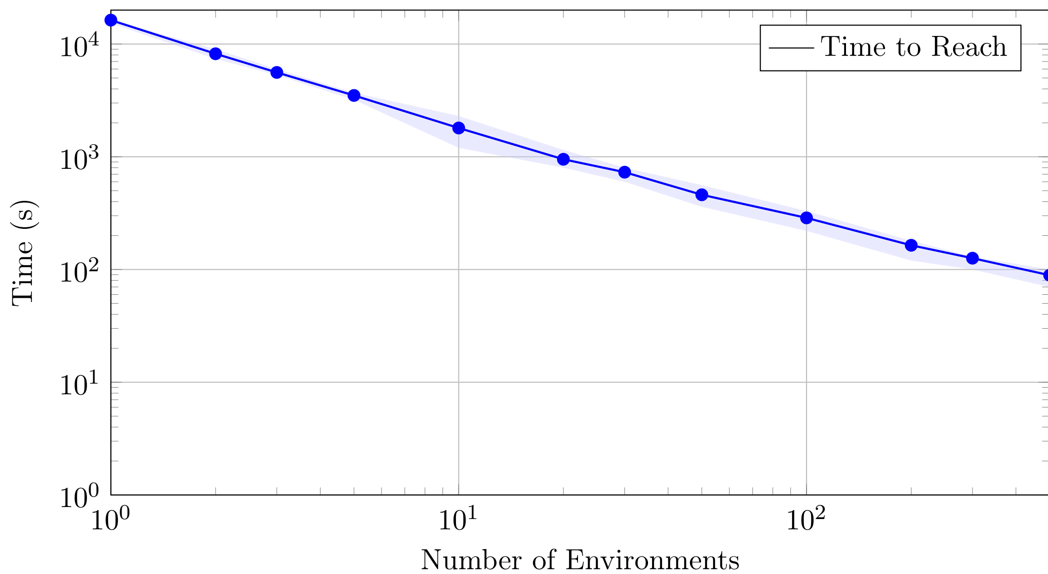


Figure 7.3: Impact of parallelization on time to reach in the ARTER environment, with variance shaded.

We can observe that increasing the number of env has a great influence on the efficiency of the algorithm with a high decrease if the time by step. The time by step is devised by up to 183 with the highest number of env (500) compared to using one single env. It can be noted that 500 was the maximum possible parallzlizeation in term of memory on the T1550.

Influence of Parallelization on the training of the ARTER environment The table 7.3 aims to evaluate the impact of parallelization on the overall performance of the PPO algorithm, which will be used in each option of our option framework. The tests were conducted on an NVIDIA T1550 GPU using the ARTER environment. Each test ran for exactly one hour with varying numbers of environments, and was repeated ten times to compute an average performance.

Number of Envs	Number of Episodes	Max Reward	Max Main Reward
1	1300	6.17	0.54
10	4200	7.18	0.63
100	17500	7.69	0.68

Table 7.3: Evaluating PPO Performance in ARTER: Impact of Parallel Environment Scaling on Reward Optimization for a training of 3600 seconds.

Conclusion This analysis revealed that parallelization significantly increased the number of episodes—by up to 13 times, within the same training duration (3600 seconds). This increase is also associated with improved rewards and overall performance, as a higher number of environments during training tends to yield the highest reward.

7.3 LLM Pipeline

Analysis of the segmentation For comparing segmentation we made a ground truth and compared the

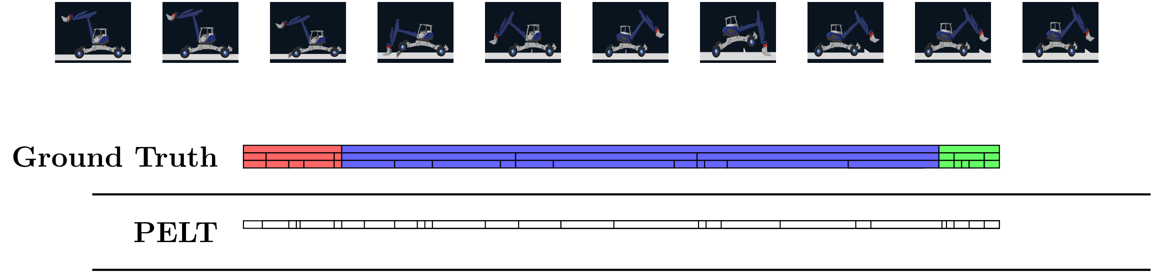


Figure 7.4: Example of the segmentation of a video

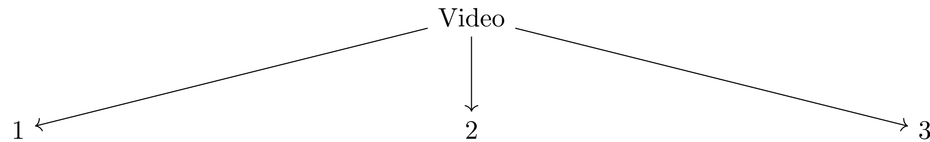
Analysis of the segmentation Unfortunately, the first thing to notice is that PELT doesn't give a hierarchical segmentation, however we can compare its output to each layer of our ground truth. We notice that it always tend to detect more change point that their actually are, which translate into low precision for each layer, respectively 10, 28 and 57 percent. However as the recall is high for high level and the F1 score tends to increase with the layer, we can conclude that PELT successfully segment the log into meaningful segment for low level, which can still be useful for the annotation.

Qualitative analysis of the annotation and reward of each segment For this experiment, we provided the annotation module with the ground truth segmentation. The reward module then received the annotated data directly. The main challenge was to extract a usable reward from the annotation: the initial reward was too complex to be used directly, as it required high-level functions that the reward generator could not produce. However, by modifying the prompt to explicitly request simpler rewards based solely on environment observations, we were able to obtain meaningful data.

This experiment was run on an A100 GPU. For clarity, this section presents only the first layer of the decomposition; a detailed breakdown of each segment's annotation and corresponding reward is provided in Appendix A.3.

Algorithm	**Precision** (%) for each layer	**Recall** (%) for each layer	**F1 Score** (%) for each layer
PELT	10 - 28 - 57	100 - 100 - 80	18 - 44 - 67

Table 7.4: Comparison of Precision, Recall, and F1 Score for Temporal Segmentation Algorithms for 10 different videos



Node	**Images**	**Segment**	**Time**	**LLM Annotation**	**Reward**
1		ARTER try to reach the obstacle	00:00–00:11	The wheel of that ARter robot turn, the robot advance, the ARter trembling a little at the end	$r = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_j$
2		Arter pass the obstacle	00:11–01:17	The Arter Robot climb over the obstacle	$r = (x_{\text{base},t} - x_{\text{base},t-1}) + \lambda \cdot z_{\text{base},t}$
3		Arter reach the goal after obstacle	01:17–01:23	The wheel of the robot turn, the robot seems to advance	$r = abs(v_{wheel})$

Table 7.5: Segment Details Referenced by Node Number

Conclusion Thanks to our qualitative analysis, it is possible to conclude that the annotations enabled the Video Annotation Module to capture the essential actions even if it lacked really important interpretive details. Therefore the module output a weak reward function that could not be used directly. As explained in 6, we attempted to run several options using PPO with the generated reward, but unfortunately, we were unable to obtain any meaningful results, even after extended training attempts.

Chapter 8

Discussion

Our empirical evaluation across PPO, Option-Critic, and HIRO confirms that hierarchical reinforcement learning (HRL) methods, despite their architectural complexity, can offer improvements in training efficiency and final performance compared to standard flat RL baselines. Notably, even relatively simple HRL approaches like HIRO demonstrated accelerated convergence and more structured exploration behavior in locomotion tasks, in the ARTER env where long-term temporal dependencies are critical (see Figure 7.1).

However, this examination further identified two notable limitations of current HRL algorithms. First, their **limited reusability** across tasks remains a key challenge : policies and learned subcomponents could not be reliably transferred to new scenarios without retraining. Second, the **lack of interpretability** of the discovered hierarchical structures, such as the internal policies and termination conditions for options-critic makes the analyzing and adaptation to new scenario of such learned behaviors difficult. These limitations provide a strong justification for developing a more principled subtask discovery pipeline, aiming to produce reusable and interpretable hierarchical decompositions.

Additionally, the experiments shows the crucial role of **parallelization** in order to efficiently scale RL algorithms. Leveraging environment parallelization (via RSL-RL [32]) significantly reduced training time by a factor of *13*, without degrading policy performance (see Table 7.3). While parallelizing the training pipeline can be complex the overall gains in computational efficiency and scalability were substantial, making this approach a valuable for the HRL framework.

Finally, we proposed pipeline aimed to enable automated sub-task discovery and hierarchical policy training by combining segmentation, annotation, and reward generation into a unified framework. Even if the segmentation proposed by PELT was not hierarchical the initial stages of the pipeline, trajectory segmentation and semantic annotation, was able to produce encouraging outcomes. However the transition from interpretable sub-tasks to useful reinforcement learning reward still remained problematic. In particular, segmentation aligned well with human intuition for the most deepened layer, and annotated descriptions were clear and contextually relevant when enhanced with domain-specific prompts even if they lack details of the overall meaning of the action. However, automatically generated rewards were too generic to capture robot-specific control requirements, resulting in a complete lack of learning progress when applied in PPO training. This bottleneck has, for now, stopped us from fully using the pipeline for complete end-to-end hierarchical skill decomposition and training, even though its modular design and potential to scale were key strengths. The results show that it's possible to automate task decomposition in HRL, but they also highlight the need for strong, well-structured reward definitions to successfully train the different policy across the hierarchical structure.

Chapter 9

Future work

While this study has made promising progress, there are still many directions to explore in order to fully tap into the potential of Hierarchical Reinforcement Learning (HRL) for robotic walking tasks.

Robust Reward Generation for Subtask Policies While the result of the automatic reward generation mechanism of the pipeline were not sufficiently aligned with robot-specific control, the segmentation and annotation part of the pipeline yielded interpretable and semantically meaningful subtasks. Future research could explored fine-tuning LLM on reward generation or improve the prompt with domain-specific priors or ask for physics-based constraints. Additionally, adding more context about the reward to generate for example adding the previous, next or even parents task could give more adapted reward. Finally in order to address the challenge of the hierarchical segmentation, other algorithm than PELT could be explored like Bayesian Online Change Point Detection (BOCPD) which is a powerful method to detect change point in time series data and could be more adapt to a hierarchical segmentation.

Transferability and Generalization of the Reward Generation Pipeline The generalization capabilities and reusability of the proposed reward generation pipeline, which leverages large language models (LLMs) to annotate and structure subtask policies within the Option framework, remain an open question. While initial results demonstrate semantic coherence and task relevance in controlled environments, its robustness across varying terrains, robot morphologies, and behavioral contexts has yet to be systematically evaluated. Moreover, the potential reusability of similar task and reward has not efficiently used and tested yet. An improvement could be to automatically identify group of similar tasks and generate one reward and so one option for this group of reward, leveraging the full capabilities of generalization of the Option-Critic architecture.

Scalable Parallelization of Pipeline Components We clearly shown that parallelized training of subtask policies significantly accelerated learning but parallelize all module of the pipeline introduced non-trivial engineering challenges. In particular, synchronizing the segmentation, annotation, and reward synthesis stages with concurrent policy optimization demands careful orchestration. Future research should explore distributed architectures that support asynchronous execution of pipeline components, enabling scalable subtask discovery and reward assignment without bottlenecking the training loop. Nonetheless the first step to improve scalability would be to establish a standardized benchmark for pipeline parallelization across diverse robotic platforms in order to identify the best practices and light the path for future deployments.

Real-World Validation of the Pipeline The pipeline seems really promising in simulations, the next step would be to test it with real-world data and then adapt it to a real environment. Real environment introduces more randomness with more noise, delay and unpredictable condition, that need to be tested. In order to smoothly bridge the sim-to-real gap in the context of the ARTER robot, first step would be to generate reliable subtask rewards in more realistic simulated environment, either by introduce more random in the simulation or by refinement through real-world feedback loops.

These future research can help make HRL more scalable, easier to interpret, and thus more useful in robotics. This could lead to smarter and more independent agents that can take on tricky tasks over long periods with little human help.

Chapter 10

Conclusion

In this project, we applied Hierarchical Reinforcement Learning (HRL) to stepping locomotion in wheel-legged robots. The main goal was to automatically break down the task into subtasks and learn the corresponding subpolicies at the same time. We tested several HRL methods, like PPO, Option-Critic, and HIRO, and found that hierarchical models generally performed better than standard flat RL. They were more efficient during training and led to stronger final policies, although they did require more training time overall.

We also looked into the influence of parallelization on training and discover that it helps scale HRL algorithms by improving computational efficiency and scalability. Although this could introduced more complexity, parallelization can become a valuable component of future HRL systems.

Furthermore, our analysis revealed two critical limitations of current HRL methods. The main limits is the lack of subtask reusability, the second is the limited interpretability of learned hierarchical structures. These challenges motivated the development of our modular pipeline that integrates trajectory segmentation, semantic annotation, and reward generation using the Option framework. While the segmentation and annotation components produced interpretable and contextually relevant subtasks, the reward generation mechanism failed to yield reinforcement signals sufficiently aligned with robot-specific control requirements, resulting in poor downstream learning performance.

Overall, this work helps to the broader goal of enabling scalable, autonomous learning in hierarchical reinforcement learning. By identifying key bottlenecks of current method and proposing a partially automated subtask discovery framework, we lay the groundwork for future research aimed at improving the generalization, interpretability, and deployability of HRL in real-world robotic systems. However new developments in robust reward modeling, transferable policy architectures, and scalable training infrastructure are needed in order to fill the gap between theoretical HRL frameworks and practical, adaptive agents capable of operating in complex, unstructured environments.

Appendix A

Additional Figures

A.1 ARTER

A.1.1 DOF

The ARTER environment action and observation is mostly based on the Degree Of Freedom DOF of the robot dtailled in A.1.

Main Part	**Number**	**Joint Name**	**Kinematic Pair**	**DOF**	**Actuation Method**	**Control Mode**	**Sensor Feedback**
Manipulator Arm	1	Cabin joint	Revolute	1	hydraulic	pos/vel	pos/vel
Manipulator Arm	1	Boom	Revolute	1	hydraulic	pos/vel	pos/vel
Manipulator Arm	1	Dipper	Revolute	1	hydraulic	pos/vel	pos/vel
Manipulator Arm	1	Telescope	Prismatic	1	hydraulic	pos/vel	pos/vel
Manipulator Arm	1	Shovel	Revolute	1	hydraulic	pos/vel	pos/vel
Manipulator Arm	1	Tilt	Revolute	1	hydraulic	pos/vel	pos/vel
Manipulator Arm	1	Roto	Revolute	1	hydraulic	pos/vel	pos/vel
Front Legs	2	Swivel	Revolute	1	hydraulic	pos/vel	pos/vel
Front Legs	2	Stabilizer	Revolute	1	hydraulic	pos/vel	pos/vel
Front Legs	2	Steering	Revolute	1	hydraulic	pos/vel	pos/vel
Front Legs	2	Wheel	Revolute	1	hydraulic	pos/vel	pos/vel
Rear Legs	2	Swivel	Revolute	1	hydraulic	pos/vel	pos/vel
Rear Legs	2	Stabilizer	Revolute	1	hydraulic	pos/vel	pos/vel
Rear Legs	2	Extender	Prismatic	1	hydraulic	pos/vel	pos/vel
Rear Legs	2	Dipper	Revolute	1	hydraulic	pos/vel	pos/vel
Rear Legs	2	Steering	Revolute	1	hydraulic	pos/vel	pos/vel
Rear Legs	2	Wheel	Revolute	1	hydraulic	pos/vel	pos/vel

Table A.1: Summary of ARTER Robot Degrees of Freedom

A.1.2 Detailled controller scheme

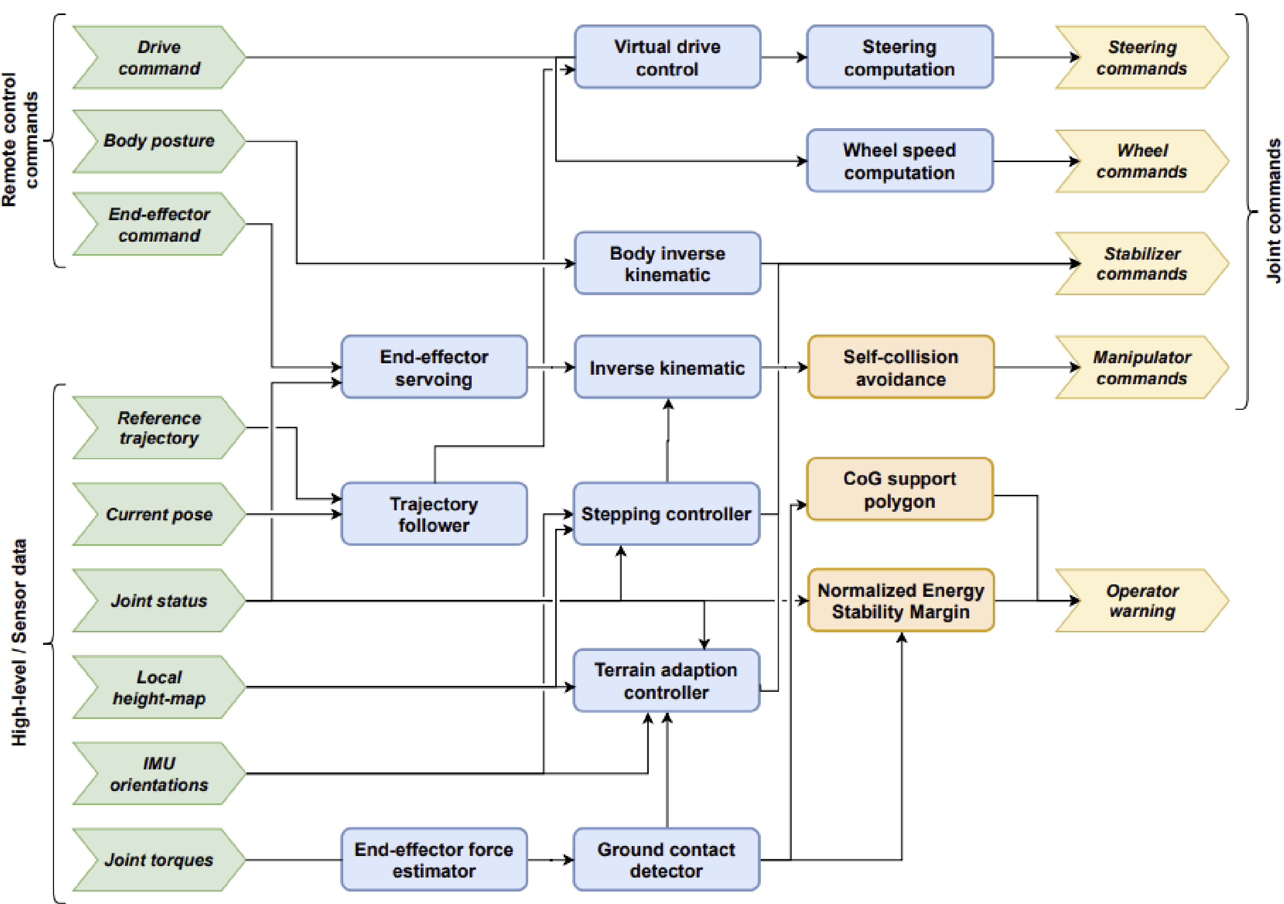


Figure A.1: Details of the mid-level controller structure. Figure from [1]

A.2 ARter environment detail

A.2.1 Action space

Type	Category	Subcategory	Number/Dim	Range
observation	Joint velocity	Wheel	4	[-1, 1]
		Cabin	1	[-1, 1]
		Boom	1	[-1, 1]
		Dipper	1	[-1, 1]
		Telescope	1	[-1, 1]
		Shovel	1	[-1, 1]
		Tilt	1	[-1, 1]
		Roto	1	[-1, 1]
		Swivel	4	[-1, 1]
		Stabilizer	4	[-1, 1]
		Extender	2	[-1, 1]
		Dipper	2	[-1, 1]
		Steering	4	[-1, 1]
		Wheel	4	[-1, 1]

Table A.2: Summary of the action of the ARTER env

A.2.2 Observation space

Type	Category	Subcategory	Number/Dim	Range
Observation	Center of mass position	x	1	[0, 1]
		y	1	[0, 1]
		z	1	[0, 1]
	Center of mass velocity	x	1	[0, 1]
		y	1	[0, 1]
		z	1	[0, 1]
	Quaternion position	x	1	[0, 1]
		y	1	[0, 1]
		z	1	[0, 1]
		w	1	[0, 1]
	Angular velocity	x	1	[0, 1]
		y	1	[0, 1]
		z	1	[0, 1]
	Joint position	Cabin	1	[0, 1]
		Boom	1	[0, 1]
		Dipper	1	[0, 1]
		Telescope	1	[0, 1]
		Shovel	1	[0, 1]
		Tilt	1	[0, 1]
		Roto	1	[0, 1]
		Swivel	4	[0, 1]
		Stabilizer	4	[0, 1]
		Extender	2	[0, 1]
		Dipper	2	[0, 1]
		Steering	4	[0, 1]
	Joint velocity	Wheel	4	[0, 1]
		Cabin	1	[0, 1]
		Boom	1	[0, 1]
		Dipper	1	[0, 1]
		Telescope	1	[0, 1]
		Shovel	1	[0, 1]
		Tilt	1	[0, 1]
		Roto	1	[0, 1]
		Swivel	4	[0, 1]
		Stabilizer	4	[0, 1]
		Extender	2	[0, 1]
		Dipper	2	[0, 1]
		Steering	4	[0, 1]
		Wheel	4	[0, 1]
	Distance	To obstacle	4	[0, 1]
		To ground	4	[0, 1]

Table A.3: Summary of the observation of the ARTER env

A.2.3 Wrapper

Wrapper name	Wrapper description
AddCameraWrapper	Allows adding one or multiple cameras to the Genesis environment to record the robot's movement.
AddCollisionWrapper	Computes collisions between two specified entities.
AddDistanceWrapper	Calculates the distance between two specified entities.
AddEntityWrapper	Adds an entity to the simulation (for example the robot or an obstacle).
AddRewardWrapper	Modify the different reward of the initial environment following the explanation in 6.1.3
ControllerWrapper	Maps actions to a remote controller using Pygame.
DiscretizeActionSpaceWrapper	Discretize the continuous action space of the environment
EpisodeLoggerWrapper	Log each episodes of the environment
HiroWrapper	Adapts the environment for compatibility with the HIRO library [33].
LoggerWrapper	Logs selected observations and actions into a CSV file.
OcWrapper	Adapts the environment for compatibility with the OC library [34].
RosWrapper	Creates a ROS node and publishes selected observations and actions to specified publishers.
RslRlWrapper	Adapts the environment for compatibility with the RSL-RL library [32].
StepLoggerWrapper	Log each steps of the environment
UnnormalizedObservationWrapper	Unnormalizes specific observations based on info dict metadata (default observations are normalized).

Table A.4: Recap of the different wrappers implemented in the environment

A.3 Complete LLM pipeline result

Node	Segment	Time	LLM Annotation	Reward
1	ARTER try to reach the obstacle	00:00–00:11	The wheel of that ARter robot turn, the robot advance, the ARter trembling a little at the end	$R = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_j$
1.1	Waiting for stabilization	00:00–00:03	The ARter robot stop	$R = -\left\lvert v_x^{\text{base}} \right\rvert$
1.2	ARTER advance to reach the obstacle	00:03–00:10	The wheel of that ARter robot turn, the robot advance	$R = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_j$
1.2.1	ARTER accelerate	00:03–00:05	The ARter robot stop	$R = -\sum_{j \in \mathcal{J}_{\text{wheel}}} \lvert \dot{q}_j \rvert$

Node	Human annotation	Human annotation Time	LLM Annotation (simplified)	LLM Reward
1.2.2	ARTER keeps it speed	00:05–00:07	The wheel of that ARter robot turn, the robot advance	$R = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_j$
1.2.3	ARTER decelerate	00:07–00:10	The wheel of that ARter robot turn, the robot advance	$R = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_j$
1.3	ARTER stops to stabilize in frint of the obstacle	00:10–00:11	The ARter robot seems to stop abruptly	$R = -\lvert v_x^{\text{base}}(t) - v_x^{\text{base}}(t-1)\rvert$
2	Arter pass the obstacle	00:11–01:17	The Arter Robot climb over the obstacle	$R = (x_{\text{base},t} - x_{\text{base},t-1}) + \lambda \cdot z_{\text{base},t}$
2.1	Front wheels pass the obstacle	00:11–00:30	The front wheel of the ARter robot pass the obstacle	$R = \mathbf{1}\{\, x_{\text{wheel_fl},t} \geq x_{\text{pass}} \;\wedge\; x_{\text{wheel_fr},t} \geq x_{\text{pass}}\,\}$
2.1.1	Deploy arm after the obstacle	00:11–00:17	The Arter robot moves its shovel	$R = \lvert \dot{q}_{\text{shovel},\,t}\rvert$
2.1.2	Push on the ground with the arm to lift the front wheel	00:17–00:21	The Arter robot moves its shovel making the robot tiping over	$R = \lvert \dot{q}_{\text{shovel},\,t}\rvert - \lambda \cdot (\lvert \text{Roll}_t\rvert + \lvert \text{Pitch}_t\rvert)$
2.1.3	Turn rear wheel to pass obstacle	00:21–00:28	The wheel of the robot seems to turn the robot advance.	$R_t = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_{j,t}$
2.1.4	The Arter robot moves its shovel making the robot trembling	00:28–00:30	-	$R = \lvert \dot{q}_{\text{shovel},\,t}\rvert - \lambda \cdot \lVert \boldsymbol{\omega}_{\text{base},\,t}\rVert^2$
2.2	Leave the arm and rotate	00:00–00:00	The Arter robot moves its shovel	$R = \lvert \dot{q}_{\text{shovel},\,t}\rvert$
2.2.1	Lift the arm above the obstacle	00:30–00:34	The Arter robot moves its shovel up	$R = abs(v_{wheel})$
2.2.2	Rotate the cabin on its base to rotate the arm	00:34–00:47	Part of the ARter robot rotate while the wheels stay on the ground	$R = abs(v_{wheel})$
2.2.3	Pose the arm below the obstacle	00:47–00:50	The Arter robot moves its shovel up	$R = \max(0,\, q_{\text{shovel},\,t} - q_{\text{shovel},\,t-1})$
2.3	Pass the rear wheel	00:50–01:17	The Arter robot moves its shovel up making the robot typing over	$R = \max(0,\, q_{\text{shovel},\,t} - q_{\text{shovel},\,t-1}) + \lambda \cdot (\lvert \text{Roll}_t\rvert + \lvert \text{Pitch}_t\rvert)$

Node	Human annotation	Human annotation Time	LLM Annotation (simplified)	LLM Reward
2.3.1	Deploy arm on the ground before the obstacle	00:50–00:51	The robot put its showel down on the ground	$R = \max(0,\, q_{\text{shovel},\, t-1} - q_{\text{shovel},\, t}) + \lambda \cdot \mathbf{1}_{\text{contact}}$
2.3.2	Push on the ground with the arm to lift the rear wheel	00:52–00:53	The robot typing over	$R = (\lvert\text{Roll}_t\rvert + \lvert\text{Pitch}_t\rvert)$
2.3.3	Turn front wheel to pass obstacle	00:53–01:15	The wheel of the robot turn trying to requilibrate the robot	$R = -\alpha \cdot \lvert\theta_t\rvert + \beta \cdot \lvert\dot{q}_{\text{wheel},\, t}\rvert$
2.3.4	Lift the arm to stop pushing on the ground and posing the four wheels	01:15–01:17	The wheel of the robot turn trying to reequilibrate the robot	$R = -\alpha \cdot \lvert\theta_t\rvert + \beta \cdot \lvert\dot{q}_{\text{wheel},\, t}\rvert$
3	Arter reach the goal after obstacle	01:17–01:23	The wheel of the robot turn, the robot seems to advance	$R = abs(v_{wheel})$
3.1	Waiting for stabilization	01:17–01:18	The robot stops	$R = -abs(v_{robot})$
3.2	ARTER advance to reach the goal	01:18–01:21	The wheel of the robot turn, the robot seems to advance	$R = abs(v_{wheel})$
3.2.1	ARTER accelerate	01:18–01:19	The wheel of the robot turn, the robot seems to advance	$R = abs(v_{wheel})$
3.2.2	ARTER keeps it speed	01:19–01:20	The wheel of the robot turn, the robot seems to advance quickly	$R = \sum_{j \in \mathcal{J}_{\text{wheel}}} \dot{q}_j$
3.2.3	ARTER decelerate	01:21–01:21	The robot stops	$R = -abs(v_{robot})$
3.3	ARTER stops to stabilize at the goal	01:21–01:23	The robot stops	$R = -abs(v_{robot})$

Table A.5: Segment Details Referenced by Node Number

The complete notation in the table are :

- R : Reward
- $\mathcal{J}_{\text{wheel}}$: Set of wheel joints
- $\dot{q}_j$: Velocity of joint j
- $\dot{q}_{j,t}$: Velocity of joint j at time t

- v_{robot} : Velocity of the robot
- v_{wheel} : Mean velocity of the four wheel
- θ_t : Orientation angle at time t
- $\dot{q}_{\text{wheel},\, t}$: Wheel joint velocity at time t
- Roll_t : Roll angle at time t
- Pitch_t : Pitch angle at time t
- $q_{\text{shovel},\, t}$: Shovel joint position at time t
- $q_{\text{shovel},\, t-1}$: Shovel joint position at time $t-1$
- $\dot{q}_{\text{shovel},\, t}$: Shovel joint velocity at time t
- $\boldsymbol{\omega}_{\text{base},\, t}$: Angular velocity of base at time t
- $x_{\text{wheel_fl},t}$: x-position of front-left wheel at time t
- $x_{\text{wheel_fr},t}$: x-position of front-right wheel at time t
- x_{pass} : x-position threshold for passing
- $x_{\text{base},t}$: x-position of base at time t
- $x_{\text{base},t-1}$: x-position of base at time $t-1$
- $z_{\text{base},t}$: z-position of base at time t
- $v_x^{\text{base}}(t)$: x-velocity of base at time t
- $v_x^{\text{base}}(t-1)$: x-velocity of base at time $t-1$
- λ : Weighting coefficient
- α : Weighting coefficient for orientation
- β : Weighting coefficient for velocity
- $\mathbf{1}_{\text{contact}}$: Indicator function for contact
- $\mathbf{1}\{\cdot\}$: General indicator function

Bibliography

[1] Babu Ajish. *Control of Robots with Hybrid Locomotion Capabilities.* Doctoral dissertation (dr.-ing.), University of Bremen, Department 3 – Mathematics and Computer Science, Unpublished doctoral dissertation. Advisor: Prof. Dr. Dr. h.c. Frank Kirchner. Doctoral colloquium scheduled for 26.08.2025.

[2] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning, 2018.

[3] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The Option-Critic architecture. *Proc. Conf. AAAI Artif. Intell.*, 31(1), February 2017.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[5] Yuxi Li. Deep reinforcement learning: An overview, 2018.

[6] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.*, 54(5), June 2021.

[7] Melike Baykal-Gürsoy. *Semi-Markov decision processes.* John Wiley & Sons, Inc., Hoboken, NJ, USA, February 2011.

[8] Martin Klissarov, Pierre-Luc Bacon, Jean Harb, and Doina Precup. Learnings options end-to-end for continuous action tasks, 2017.

[9] Matthew Riemer, Miao Liu, and Gerald Tesauro. Learning abstract options, 2019.

[10] Khimya Khetarpal and Doina Precup. Learning options with interest functions. *Proc. Conf. AAAI Artif. Intell.*, 33(01):9955–9956, July 2019.

[11] Shangtong Zhang and Shimon Whiteson. Dac: The double actor-critic architecture for learning options, 2019.

[12] Martin Klissarov, Akhil Bagaria, Ziyan Luo, George Konidaris, Doina Precup, and Marlos C. Machado. Discovering temporal structure: An overview of hierarchical reinforcement learning, 2025.

[13] Christian Daniel, Herke van Hoof, Jan Peters, and Gerhard Neumann. Probabilistic inference for determining options in reinforcement learning. *Mach. Learn.*, 104(2-3):337–357, September 2016.

[14] George Konidaris and Andrew Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[15] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning, 2017.

[16] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Near-optimal representation learning for hierarchical reinforcement learning, 2019.

[17] Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning (ICML)*, pages 361–368, San Francisco, CA, 2001. Morgan Kaufmann Publishers Inc.

[18] Özgür Şimşek and Andrew G. Barto. Skill characterization based on betweenness. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 21 (NIPS 2008): Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, pages 1497–1504, USA United States, 2009. Curran Associates, Inc.

[19] Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning (ECML)*, pages 295–306, Berlin, 2002. Springer-Verlag.

[20] Özgür Şimşek, Alicia P Wolfe, and Andrew G Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning - ICML '05*, New York, New York, USA, 2005. ACM Press.

[21] Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. A laplacian framework for option discovery in reinforcement learning, 2017.

[22] Bram Bakker and Jürgen Schmidhuber. Hierarchical reinforcement learning with subpolicies specializing for learned subgoals. pages 125–130, 01 2004.

[23] Sainbayar Sukhbaatar, Emily Denton, Arthur Szlam, and Rob Fergus. Learning goal embeddings via self-play for hierarchical reinforcement learning, 2018.

[24] Karol Hausman, Jost Tobias Springenberg, Ziyu Wang, Nicolas Heess, and Martin Riedmiller. Learning an embedding space for transferable robot skills. 2018.

[25] Joshua Achiam, Harrison Edwards, Dario Amodei, and Pieter Abbeel. Variational option discovery algorithms, 2018.

[26] John D. Co-Reyes, YuXuan Liu, Abhishek Gupta, Benjamin Eysenbach, Pieter Abbeel, and Sergey Levine. Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings, 2018.

[27] Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning, 2024.

[28] Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning, 2024.

[29] Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. Auto mc-reward: Automated dense reward design with large language models for minecraft, 2024.

[30] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *ICLR*, 2024.

[31] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[32] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 91–100. PMLR, 2022.

[33] Kandai Watanabe and Matt Strong. hiro_pytorch: An implementation of data-efficient hierarchical reinforcement learning (hiro) in pytorch. https://github.com/watakandai/hiro_pytorch, 2025. Accessed: August 26, 2025.

[34] Laurens Weitkamp. option-critic-pytorch: Pytorch implementation of the option-critic framework. https://github.com/lweitkamp/option-critic-pytorch, 2025. Accessed: August 26, 2025.