Alma Mater Studiorum · Università di Bologna

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Second Cycle Degree in Artificial Intelligence

Development of a Deep Reinforcement Learning System for Adaptive Movement of Hybrid Robots

Supervisor:
Chiar.mo Prof.
Andrea Asperti

Student:
Andrea Rossolini

Co-supervisor (external):
Alex Ramirez-Serrano
(University of Calgary,
Alberta, Canada)

Extraordinary Session
Academic Year: 2021/2022

Code available at: (algorithm + Cricket 3D models)
https://github.com/AndreaRoss96/gym-cricket-robot and (Cricket 3D model construction) https://github.com/AndreaRoss96/CricketRobot-3Dmodel

Summary

This thesis is the result of a research and development project carried out in collaboration with the University of Calgary (Alberta, Canada), in particular with the robotics laboratory "UVS Robotarium Lab" and the company "HEBI Robotics" that collaborates with them, under PhD Alex Ramirez-Serrano's supervision. The project consists of research and development of a deep reinforcement learning (RL) system to make a multilegged robot, with different degrees of freedom, able to adapt to different terrain. Recent advances in deep RL have created incredible opportunities for intelligent automation, especially robotics, and reached outstanding results. However, current deep RL algorithms predominately specialize in a narrow range of tasks. They are sample inefficient and ineffective when the agent needs to tackle extremely complex environments, or when the agent possesses unique characteristics. Compared with the conventional tracked/wheeled robots, the reconfigurable hybrid robots can alter their structures and configuration to better adapt to the various, complicated and completely unstructured environments. Considering the concept of reconfigurability, to overcome the weakness of tracked robots with two parallel tracks. A reconfigurable hybrid robot can locomote autonomously, using diverse abilities (e.g., walk, roll, climb, crawl, etc.) with different modalities (e.g., craw with hands and feet or with elbows and knees), in unstructured environments. To move around with ease, it needs to adapt to a possibly infinite set of different environments and, at the same

time, select a specific locomotion skill. This is, indeed, extremely complicated. This project aims to deal with these limitations by creating a system able to let the robot learn how to adapt to different kind of environments singularly and, therefore, create a previous knowledge that the robot could use to overcome different situations and let it selects the most useful locomotion technique to achieve the given task. This thesis describes the main characteristics of the proposed algorithm and the technical aspects behind the code. The algorithm used to implement the solution is based on the famous Deep Deterministic Policy Gradient (DDPG). In this thesis, I show the implementation of this algorithm, already applied to solve the problem of locomotion of agents within an environment, with some changes to its architecture and applied to a highly reconfigurable hybrid robot. These changes affect the neural network used by the agent since in this solution the robot uses stereo-cameras to investigate its surroundings, and a pose to be achieved, not only defined by its position in space, but also by the position of its limbs and joints. The final goal is to build a system that, given a final pose, allows the hybrid robot to develop a policy and adapt to a given terrain. The developed system will be a part of a larger project, a research theme within my host university, that aims to enable the robot to learn how to adapt and transition between diverse states and locomotion modalities (e.g. rolling, crawling, climbing, bipedal and quadrupedal walking etc.) while maintaining its balance. Moreover, the work also includes a part of modelling and design of the code and simulation. The robot used to carry out this project has never been used in this field by the university lab. Since there was no framework or previous research, the coding and simulation parts were developed from scratch. In conclusion, I provided the initial steps for the University of Calgary's research in this field. The results obtained show that the Deep RL system developed in this thesis can quickly find a policy that obtains a good reward on basic and polished terrains. However, the system struggles when it faces complex terrains, but is still able to find an appropriate policy that

allows him to move adequately in the terrain, leaving room for improvement.

This thesis is divided into four parts: Firstly, a discussion on the general structure of hybrid robots and the specific robot used in this research, its unnatural locomotion capabilities, and how it works in the virtual environment. Secondly, the main features of the DDPG algorithm, with emphasis on neural networks, differ from the originals. Then, the formulation and explanation of the unique reward function. In conclusion, we analyze the results obtained and the future developments. These topics are explained and discussed in 9 chapters:

- **Chapter 1 - Introduction**: Introduction to intelligent automation, reinforcement learning and deep reinforcement learning.
- **Chapter 2 - Literature Review**: Review of the sources and literature studied during the develop of this thesis. An overview on the research in reinforcement learning applied to robotics.
- **Chapter 3 - Hybrid Multi-locomotion Robot System**: Overview on different type of hybrid multi-locomotion robots and a description of the robot named "Cricket" used as subject of this thesis.
- **Chapter 4 - The Problem of Locomotion in Unstructured Terrains**: Explanation of the problem and the proposed solution
- **Chapter 5 - Environment**: Focus on the development of the characteristic of environment framework used to train the Cricket robot.
- **Chapter 6 - Reward Function**: Explanation of the reward function and why such choices were made.
- **Chapter 7 - Actor and Critic**: Description of the Agent and Critic interaction and implementation.

- **Chapter 8 - Implementation**: Implementation and delineation of the technical characteristics.
- **Chapter 9 - Conclusion**: Conclusion and future works.
- Appendix: Delineate the code structure, technologies used and other information.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Advances in robotics and computing technologies are giving rise to a new generation of "intelligent" automation technologies that can transform various needs such as healthcare, defence, transportation, and emergency among others. Automation and robotics are the main core of the above needs, from processing and part transfer in factories to assistive robots in hospitals, unmanned aerial drones and vehicles in agricultural, defence, and rescue fields [1]. Many robotic operations in the industry are, for now, constrained to repetition and planned tasks performed within structured environments. This leaves a whole swath of more complex tasks with high degrees of uncertainty and dynamic environments [2], that are difficult or even impossible to automate. Examples include safe working in environmental disasters [3], selective weeding in agriculture systems [4], and secure management in deep-sea operation.

Adaptability is among the key characteristics of robotic automation systems in an unstructured and unpredictable environment. In recent years the development of advanced robotic control techniques has led to the emergence and growth of adaptive

automation tools, such as collaborative robots (cobots), and learning capabilities, that allow them to operate in hazardous jobs, sustain complex tasks and work for a long duration in assembly lines. This type of robot works in environments where unplanned events are uncommon. Indeed, such robots have two major limitations: firstly, an industrial robot requires robotic control engineers to program the trajectory and sequence of their desired motion (usually in a specific programming language); secondly, they can complete a limited set of tasks that require position control and pre-defined trajectory and sometimes these tasks are excessively challenging. To address these limitations, artificial intelligence plays a key role. Nowadays, many robots can perform high-level tasks and can also decide and learn from experience in various situations. Engineers and scientists can design efficient control algorithms and architectures that enable robots to modify their behaviours to deal with uncertain circumstances and automatically improve themselves over time, with minimal or no need for reprogramming or human intervention in the robot's behaviour. The scientific community, active in the development of systems capable of dealing with environments where unpredictable events can be numerous, is growing.

The UVS Robotarium Lab at the University of Calgary is born to develop industrial robots and unmanned aerial vehicles (UAVs), humanoid robots and other robotic systems able to locomote under human control or with robot planning algorithms. Recently, The lab extended its research to the field of autonomous locomotion and artificial intelligence applied to robots. Indeed, the problems addressed in this thesis pave the way for a new lab research topic, and so, it's part of a larger project. This larger project consists of the design and autonomous control of multi-locomotion and reconfigurable robots having an advanced and broader range of capabilities in comparison with traditional legged, ground rovers, and aerial systems (drones). This project uses a new robot concept design with unique multi-locomotion capabilities

which enable it to locomote in different styles including bipedal and quadrupedal walking, locomotion, climbing (e.g., ladders rock walls), rolling (like tracked vehicles) among others. To realize effective locomotion, the lab considers the problems associated with kinematics and dynamics, improving stability, energy consumption, velocity, and manoeuvrability. They aim to solve problems related to locomotion transition by inter-connect different types of locomotion into a unified locomotion architecture that would enable robots to adapt to the perceived environment by selecting the best locomotion style for a given terrain/space situation. The overall goal is to fully realize hybrid robots by developing the required enabling mechanisms such as advanced locomotion techniques and robot self-adaptation to its surrounding in an autonomous fashion.

This thesis presents a sub-project that, given the terrain and optimal pose, aims to develop an adequate policy to make the robot adapt to the given terrain. To realize this, I applied a deep reinforcement learning algorithm, called Deep Deterministic Policy Gradient (explained below), adapting the architecture of the original algorithm to the robot characteristics and the problem under consideration. So, this project modifies the original algorithm by implementing an additional neural network, that extracts the terrain features, and specifies the final state (so not just the location) that defines the optimal position, joint angles and orientation characteristics of the robot.

In recent years, reinforcement learning (RL) algorithms achieved terrific success in enabling intelligent control of robotics manipulators in simulated (and physical) environments [5]. RL allows the agent to learn behaviours and adapt by trial-and-error and generalize the learned policies to unseen variations of the same task without changing the model itself [6]. There are no studies that show the implementation of a generalized AI that allows a terrestrial agent to autonomously adapt to unstruc-

tured terrains like an animal would do. This project aims to divide into sub-scenarios those that are critical aspects of a movement in an unstructured terrain and develop a system capable of generating an exclusive policy for these scenarios. In this way, it will be possible to save the various policies and keep them available in the future. Consequently, the agent can retrieve these policies when it encounters a criticality similar to one it has already faced, exactly as a biological memory would behave, to use them when it faces a complex path. This prior knowledge requires a reinforcement learning framework and a way to practically save the computed policy. The policy selection process is still being studied by the University of Calgary. To date, the idea is to separate different level of hierarchies and use an approach where an on-policy method train all levels of the hierarchy jointly.

Besides I would be interested, this thesis this thesis covers only with the lowest level of the algorithm.

1.1 Fundamentals of Reinforcement Learning

The goal of reinforcement learning is for the agent to develop an optimal, or near-optimal, strategy that maximises a user-supplied reinforcement signal that accumulates from the immediate rewards; usually, it is a function called ”reward function“. This appears to be a mechanism that occurs in other biological processes. Reinforcement learning is based on the idea that an agent interacts with its surrounding (or environment) in distinct time steps. At each time t, the agent is represented by a state s_t, chooses an action a_t, and receive a reward r_t. The action selected (from a space of possible actions) produces an effect on the environment which evolves to a new state s_{t+1}. The action effects get evaluated and this produces a reward r_{t+1} associated with this transaction. In conclusion, the agent aims to maximize (or minimize,

depending on the problem) the expected cumulative reward.

To approximate a basic algorithm of reinforcement and formulate it in mathematical terms we can make the following definitions: Given a set of states S and a set of action A, the agent may choose ant action $a \in A$ that is available in state $s \in S$. The probability that the environment evolves into the new state s' is influenced by the action. Hence, we can write $P(s_{t+1} = s'|s_t = s, a_t = a)$ as the probability of transition (at time t) from state s to state s' given the action a. Each transition from s to s' given a, corresponds to an immediate reward $R_a(s, s')$. The agent attempts to learn a policy $\pi(a|s)$, that maps actions to states, to maximise the sum of immediate rewards. In this case, the algorithm only has access to the values of $P(s'|s, a)$ through sampling. The previous formulation is a Markov Decision Process (MDP), a time-discrete stochastic control process, used in different fields such as optimization problems, decision making, economics, robotics and others. Figure 1.1 shows a representation of the standard reinforcement learning principle, where the agent, in a given state, produces an action and the environment evaluates it.

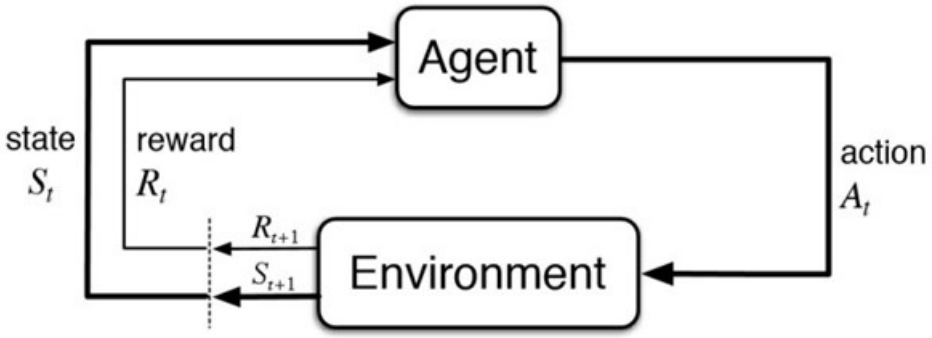


Figure 1.1: Main principle of reinforcement learning.

Model-based and Model-free

In Reinforcement Learning, the term "model" refers to the different dynamic stats of an environment and how these states lead to a reward (not to be confused with the policy, which refers to the strategy used to determine what action to take, based on the current state). In RL, it is possible to divide the approaches into two macro-groups: "model-based" and "model-free". In the first one, the agent uses a mode to represent the domain, this gives the agent the chance to "plan" the future actions. So a model-based approach builds a predictive model of the domain that helps it to choose the best action. This group typically includes dynamic programming algorithms such as Dyna Algorithm or algorithm for policy Iteration. On the other hand, model-free approaches prefer to learn a control policy, regardless of the response of the environment to the agent's actions. So, in simple terms, a model-free approach is more of a "trial-and-error" approach. This group includes algorithms such as SARSA, Q-Learning, Gradient Monte Carlo, Actor-Critic and many others. Model-free methods can have an advantage over more complicated methods when the difficulty of developing a sufficiently realistic environment model is an underlying barrier to solving a problem.

On-Policy and Off-Policy

At any time, an agent's behaviour is defined in terms of the policy. The policy π specifies the probability that an action a is taken in a state s. As already said, the goal is to find a policy that maximizes the reward. In RL there are two approaches called "Off-policy" and "On-policy". An Off-policy agent learns the value of the optimal policy regardless of the agent's actions. Hence, Off-policy algorithms evaluate and improve a policy that is different from the policy that is used for the selection of actions. An example of an Off-policy algorithm is *Q-learning*. An On-policy agent

evaluates and/or improves the policy that is used to select actions. For the On-policy algorithm, there is SARSA.

Off-policy methods result in more effective continuous exploration, as an agent learns other policies than it can be used for exploration while learning the optimal policy. While an On-policy approach risks learning a sub-optimal policy.

Richard Sutton's book "Reinforcement Learning: An Introduction" gives an outstanding overview of reinforcement learning [7].

Actor-Critic Method

The principle of actor-critic learning consists of a combination of policy learning and value learning. As an agent takes actions and moves through an environment, it learns to map the observed state of the environment into two possible outputs: the policy function and the estimated reward. As for the definition of policy function, each action in the action space has a probability value. This value is computed by the actor. On the other hand, the critic computes the estimated reward, which consists of the sum of all rewards that the agent expects to receive in the future. Agent and critic learn to fulfil their responsibilities in such a way that the actor's proposed actions optimise and maximize the reward.

1.2 Fundamentals of Deep Reinforcement Learning

Deep reinforcement learning (DeepRL) adds deep learning into the system, enabling agents to make decisions based on unstructured input data without the need for manual state space engineering. DeepRL algorithms can process large amounts of data,

such as images or natural language, and determine what steps to take to achieve a goal. Many real decision-making issues have high-dimensional MDP states that are difficult to address by classic RL algorithms. DeepRL has, in part, garnered the attention of the scientific community thanks to the work of Mnih et al. where the Deep Q-Network (DQN) architecture [8]. The architecture was able to learn alternative policies for a range of different video games, using only pixels as input data. To generalise gathered experience to unseen states, and untested actions, the $Q(s_t, a_t)$ function is represented using a deep neural network. This off-policy algorithm demonstrated above human-level performance on a set of classic Atari 2600 video games. The DQN algorithm has been heavily studied, and various improvements have been proposed such as Double Q-Learning (DDQN). Other popular DeepRL algorithms are: Agent57 (which overcomes the results obtained by DQN), Trust Region Policy Optimization (TRPO), and Proximal Policy Optimization (PPO) have also been able to work directly on high-dimensional input spaces using deep neural networks.

Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) [9], the deep learning alternative of the deterministic policy gradient with function approximation [10], is one of the most popular off-policy actor-critic algorithms. It applies a Q-function estimator to allow off-policy learning and a deterministic actor to maximize the Q-function. It uses off-policy data and the Bellman equation, described in [11], to learn the Q-function and uses the Q-function to learn the policy. Moreover, the results presented by Popov et al. [12] confirm the fact that DDPG is successful for the cases with binary reward functions. This approach has limitations: it is expensive in terms of resources and needs powerful hardware to show the best of itself. The applications of the DDPG are not related just to robotics, as it is shown in [13], this is to say this approach is

reliable even in dynamic fields such as artificial intelligence.

This thesis proposes deep neural networks, in combination with a reinforcement learning algorithm, and a unique reward function to compute an effective policy to allow an agent to adapt to different terrains. This allows for the design of control policies on high-dimensional character state description (48D) and an environment state that consist of three-dimensional terrain, with different dimensions regarding the type of the terrain (sloping, rocky, rough, smooth). A parameterized action space is provided (40D) that allows the control policy to operate at the level of bounds, leaps, steps, rolls, pitches, and so on. A modified version of the Deep Deterministic Policy Gradient (DDPG) algorithm is introduced which, in its vanilla version, does not consider the environment in which the agent moves. This DDPG develops n individual control policies and their associated value functions, which each then specializes in regimes of the overall motion, and executes the policy associated with the highest value function, in a fashion analogous to Q-learning with discrete actions.

1.3 Research contribution

One of the primary goals of robotics is to make life easier for humans and to enable us to achieve goals without limitations. To achieve this goal it is necessary to find ways to equip robots with artificial intelligence and make them autonomous. The research towards more and more "generic" intelligence continues to advance without interruption, but it is an extremely vast field.

The Unmanned Vehicle Systems (shortly UVS Robotarium Lab[1]) facility is a research laboratory created for the development of systems targeted for the enhancement of mobile robotics including artificial intelligence and formal techniques ap-

[1]https://www.uvs-robotarium-lab.ca/

plicable to aerial, ground, underwater as well as humanoid robotics. The laboratory currently has different projects where students and researchers apply their knowledge. The project on which I have collaborated consists of the development of a system to adapt hybrid robots (see section 3) to unstructured environments. The realization of such a system consists of a hierarchical learning system that recognizes or attributes certain characteristics to an environment and uses "past experiences" to adapt to it.

Hierarchical reinforcement learning (HRL) divides a reinforcement learning problem into a hierarchy of sub-problems or sub-tasks, with higher-level parent tasks invoking lower-level child tasks as if they were primitive actions. There may be numerous levels of hierarchy in a decomposition. A learned policy is extremely difficult to apply to a new problem. Sub-behaviors, on the other hand, can be transferred to tackle new issues in similar situations once acquired. Learning sub-behaviours should be weighted across several tasks rather than a single one. The ability to reuse sub-behaviours is a critical step toward agents capable of continuous learning. The work presented in this thesis is at the "lowest" level of this hierarchical learning i.e. primitive actions. The benefit of hierarchical decomposition is that it reduces computational complexity because the overall problem can be represented more compactly and reusable subtasks can be taught or provided independently. While the solution to an HRL problem is optimal given the hierarchy's restrictions, there are no guarantees that the deconstructed answer is an ideal solution to the original reinforcement. The robot used to carry out this project has never been used in this field; hence, there is no framework or previous research, and the programming and simulation part as well as the idea of the hierarchical system and deep reinforcement learning system have been completely developed from scratch.

This thesis presents the work of building a framework for creating the lowest-level of this hierarchical system. The presented approach allows to "feed" the robot with the terrain and an optimal pose and computes an appropriate policy, regardless of

the starting position, to reach the optimal posture as efficiently as possible, avoiding collisions with the surroundings, collisions with itself (e.g. one limb hitting another, or hitting the main body) and trying to avoid wasting energy and time. Therefore, to use this framework, the robot must be able to recognise its surroundings (through sensors or cameras). Moreover, it is necessary to know the optimal final position; hence, the level of joint rotation, the centre of mass position, limb positions, and so on.

The results explained in section 8.4 show that the Deep RL system developed in this thesis can quickly find a policy that obtains a good reward on basic and polished terrains. In fact, it can be understood from the results how the robot is able to stand up in flat and relatively flat terrain, deal with sloping terrain and overcome steps. However, the system struggles when it faces complex terrains (as for the steps), but is still able to find an appropriate policy that allows him to move adequately in the terrain, leaving room for improvement.

Despite the numerous limitations of the approach presented, we believe it can open many interesting directions for exploration. This approach can be used for a different kind of robot manipulation, for example to grab specific objects. The system can integrate diverse skills that would enable a character to perform more challenging tasks and more complex interactions with their environments.

Chapter 2

Literature Review

The goal of the thesis is to build a system that generates policies for the cricket robot and makes it able to autonomously move through different environments. Therefore, create an "adaptive memory" for the robot to use when similar terrain conformations are encountered [14]. A reinforcement learning algorithm able to handle variables in continuous action space is required to reach this goal. Several reinforcement learning algorithms have been studied to solve this type of problem. For this, the first step is to identify which of the approaches is better between an on-policy and off-policy algorithm. On-policy algorithms are used when there is a need to optimize the value of an agent that is exploring. Popular deep reinforcement learning algorithms that perform this optimization are TRPO [15] and PPO [16]. Another popular on-policy algorithm is the A3C [17]. An example of the application of the last algorithm can be read here [18]. This paper from 2019 applies a version of the Asynchronous Advantage Actor-Critic (A3C), to make an agent solve the dungeon in the famous video game "Rogue". On the other hand, off-policy algorithms are excellent when the exploration is limited, for example, movement predictions in robotics. For the

proposed robot an offline learning algorithm was selected to use the experience to develop a policy, without any online interaction. Although offline RL strategies can learn from prior information, there's no clear preparation for making different plan choices, from demonstrating design to calculation hyperparameters, without really assessing the learning approaches online [19]. In offline RL algorithms, it is essential to learn the value function in addition to the policy because the policy will be updated by the information received from the value function. The actor-critic method presented in this thesis has this as the core idea [20]. Actor-critic consists of two models, an actor, and a critic. The critic approximates the action-value function or state-value function, which is then used to update the actor's policy parameters [21]. The number of parameters that the actor needs to update is small compared to the number of states. Therefore, a projection of the value function is computed onto a low-dimensional subspace spanned by a set of basic functions. It is demonstrated by the parameterization of the actor. One of the extensions of the original actor-critic method is defined based on the difference between the value function and a baseline value, which is known as Advantage Actor-Critic (A2C) method [22]. The literature reports many interesting studies such as the *Soft Actor-Critic* (SAC) algorithm [23], an efficient algorithm where the actor aims to train a policy that is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. Increasing entropy encourages better exploration, which can speed up learning later on; therefore keeps the policy from getting to a bad local optimum too soon. Another recent algorithm is the *Distributional Soft Actor-Critic* (DSAC) algorithm [24]. This is an off-policy RL method for continuous control setting, to improve the policy performance by mitigating the over- or under-estimation of the Q-value estimates. Recently, papers have been published framing model-based problems where the development of deep reinforcement learning algorithms and policy search algorithms is critical [25].

Chapter 3

Hybrid Multi-locomotion Robot Systems

When compared to the field of industrial robot arms, or robot manipulators, which has already achieved significant success in industrial manufacturing, mobile robotics is a relatively new field. All major robotics organisations, such as the International Federation of Robotics and the European Robotics Research Network, predict that the global market for mobile service robotics will grow dramatically over the next two decades, surpassing the market for industrial robotics in terms of units and sales [26].

A mobile robot, in contrast to robot manipulators that are normally attached to the ground, may move around its environment to complete its work. As a result, the primary problem in mobile robotics is locomotion.

"How should a robot move? Why a particular locomotion style (e.g. walking,

rolling, swimming, flying etc.) is more appropriate than another alternative locomotion for a given mission?" [27] are some of the questions a robot must answer while traversing any homogeneous or heterogeneous (rough or smooth, static or dynamic, etc.) environment.

Mechanisms for exerting pressures and torques on the environment, a sensing system for seeing the environment, and a decision and control system that governs the robot's behaviour to achieve the intended objectives are all common components of mobile robotic systems.

Wheels/tracks and legs are the two most prevalent mechanical systems for movement in terrestrial or ground robots [26], [28]. Powered wheels can achieve incredible efficiency in terms of velocity and energy on flat, hard terrains, but struggle on uneven, bumpy, and soft terrains. The articulated leg, on the other hand, is the most basic bionic method that can be utilised to traverse a wide range of challenging terrains. The efficiency of legged locomotion systems, on the other hand, is determined by leg and body mass rather than the robot's interaction with the terrain. However, legged locomotion has a higher mechanical complexity than wheeled locomotion, owing to the higher degrees of freedom (DOF) required for walking locomotion [29].

Robots, on the other side, are frequently required to work in settings that necessitate the use of wheels and legs. Operation conditions for robots used in Urban Search and Rescue (USAR) include both flat grounds such as concrete/wooden floors and difficult terrains such as steps/stairs and unstructured surfaces such as rubble. Legged robots are good in rocky terrain but inefficient on flat land, and any form of walking gait that must be used requires sophisticated control. On flat ground, wheeled/tracked robots are extremely efficient, although, in rough terrain, they are inefficient and, in certain situations, incapable of navigating.

Hybrid solutions offer a fascinating compromise by combining the versatility of legs with the efficiency of wheels/tracks. Four of the top five teams' robot designs in

the 2015 DARPA Robotics Challenge finals (DRC) integrated legged and wheeled/-tracked locomotion. DRC was a ground robot competition held from 2012 to 2015, aimed at accomplishing complicated tasks in dangerous, degraded, human-engineered environments. The DRC followed the DARPA Grand Challenge and DARPA Urban Challenge. DRC 2015 was mentioned because it inspired and originated the final design of the robot taken into account for this thesis. In addition, subsequent competitions have featured robots with some different and more advanced features. Robots with several locomotion modes are referred to as “Hybrid Robots” in the context of this thesis. Hybrid mobile robots appear to be superior to non-hybrid mobile robots for difficult tasks, while this is not guaranteed.

Because it improves environmental adaption, locomotion variety, and operational flexibility for robots, multimodal locomotion is gaining interest in robotics research [30]. Transitions between flying and terrestrial locomotion modes [31], swimming and terrestrial locomotion modes [32], and transitions between different terrestrial locomotion modes [29] have all been studied previously. The bulk of proposed hybrid robots in the previous decades have been wheel/track-legged systems [26] due to their superior locomotive efficiency (in terms of velocity and energy) and hard terrain navigation abilities [33]. Unlike modular robots, [34], where reconfiguration control refers to a method for transforming a given robotic configuration to the desired configuration through a series of module detachments and reattachments [35], locomotion mode transition control of hybrid robots typically refers to the decision-making process that selects the most appropriate locomotion mode from the robot’s multi-locomotion modes. Locomotion transition can be accomplished via "supervised autonomy" [36], in which operators make transition decisions, or autonomously, in which robots switch between locomotion types depending on pre-determined criteria and procedures [37]. The supervised locomotion transition control requires continuous human-robot interaction, which is not always available or reliable [38], especially

when robots are used in confined and complex environments like those found in USAR tasks and other applications where operators lack complete situational awareness to make effective decisions.

3.1 Cricket: a Hybrid Multi-Locomotion Robot

The research reported in this document is focused on the reconfigurable hybrid robot shown in figure 3.1. The picture 3.1a shows the new robot based on the prototype realized by and developed by HEBI Robotics in joint coordination with the University of Calgary. The robot has been developed at the University of Calgary since 2009 [39] and built by the students and researchers at the *UVS Robotatium Lab*[1]. This primitive version is shown in picture 3.1b. As a result of a collaboration between the university and the company *HEBI Robotics*[2], the robot and its design have been upgraded to a more sophisticated version shown in figure 3.1a. If desired, the original robot's tracks can be replaced with wheels or any alternative foot layout

(a) Cricket Robot in a real world environment (b) Cricket Robot's old version

Figure 3.1: Real Cricket Robots pictures

[1]https://www.uvs-robotarium-lab.ca/
[2]https://www.hebirobotics.com/

The characteristics of the reinforcement learning system, explained in the following chapters, take into account the original robot design. This robot is capable of walking in both bipedal and quadruped styles, as well as crawling, rolling, (with the asset of 4 tracks and feet), and climbing. This robot has been nicknamed "Cricket", due to the initial geometric configuration similar to an insect (not shown here). In this section, the robot is to be described from a locomotion perspective point of view. As in the figures in the following pages, the robot's locomotion system resembles a hybrid quadruped robot with a main body and four rotational joints per leg. Figure 3.2 shows the leg components, starting from the joint closer to the main body they are named: *(1) shoulder joint 1, (2) shoulder joint 2, (3) knee joint, (4) ankle joint.*

All joints can rotate 360 degrees except for the *shoulder joint 1* which can span $\pi/2$. This configuration allows Cricket to reach 16 degrees of freedom (DOF). Besides, the robot is competent to perform advanced maneuvers not conceivable by conventional tracked vehicles, such as strolling in diverse walk designs and step climbing [40] (see figure 3.3a and 3.3b) but is also able to assume positions not common in nature, as in the figures 3.4a and 3.4b. In addition to the four rotational joints, each leg highlights a derivable track encompassing the furthest leg section, which permits the robot to drive similar to skid-steer tank robots or variation when repositioning the track concerning the robot's body by the leg motions (see figure 3.5a, 3.5b, and 3.6). The robot's kinematics also enables it to reconfigure and perform movements and locomo-

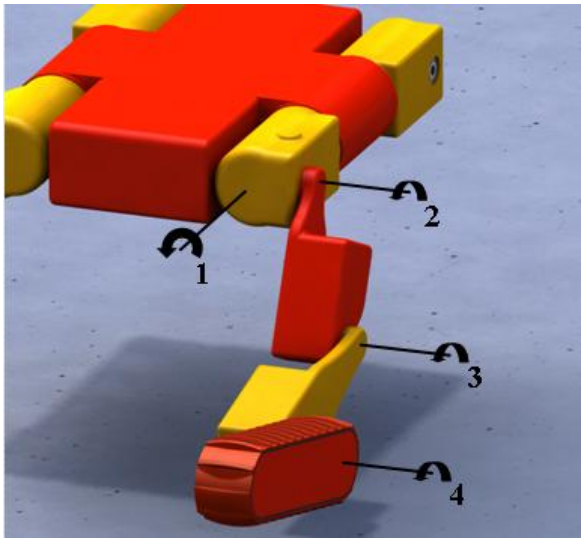
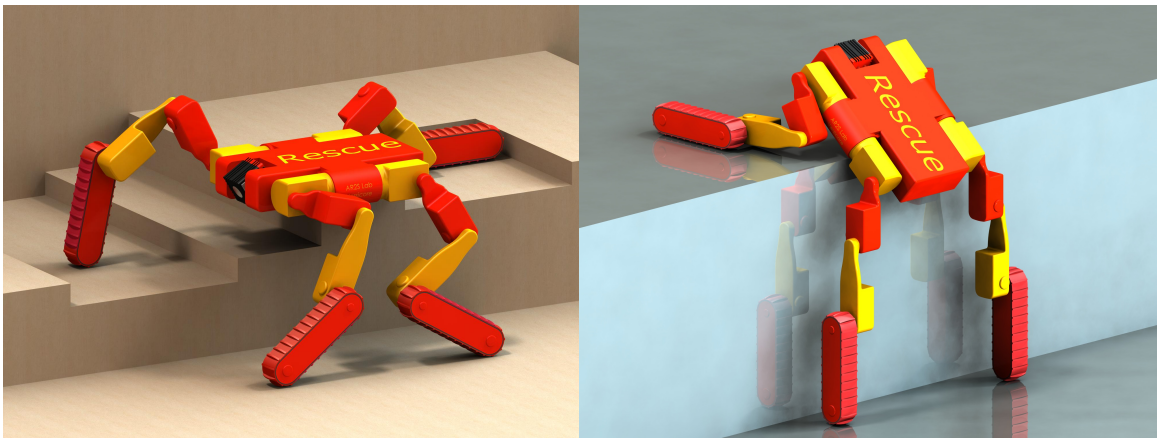


Figure 3.2: Cricket robot joints configuration

tion not seen in the natural world (see figure 3.7a and 3.7b). The motion framework of the Cricket robot empowers two primary shapes of development, strolling for navigating complex harsh territory, and rolling utilizing treads or wheels for effective travel on open semi-flat landscapes. Other manoeuvres that the robot can execute include vertical climbing activities and overcoming steps. In Figure 3.3a, the robot moves in a hybrid movement fashion combing waking and rolling at the same time to save energy, while in figure 3.3b it raises its body to overcome a large step. Furthermore, the robot is capable to perform sophisticated manoeuvres not possible by traditional tracked vehicles such as walking in diverse gait patterns and ladder climbing (figure 3.4b) Figure 3.4a illustrates a case of a challenging rough landscape and obstacle arrangement that the robot is hypothetically able to traverse. Figures 3.5a and 3.5b show examples of different leg configurations that can be executed during rolling locomotion. Figure 3.6 illustrates a case when the front left leg is changing its motion mode from rolling to walking. From these locomotion illustrations, it can be seen that the robot can raise or lower its centre of mass (COM) to improve its stability, dynamics, etc, amid its movement. Other capabilities incorporate tilting of the body as illustrated in Figure 3.4a and 3.4b to manoeuvre its body through challenging spaces. The tracks at the end of the legs allow the robot to move, and also help to overcome steep terrain and obstacles. A set of video stereo-cameras placed on the main body allow the robot to represent its surroundings as a 3D map. This representation will then be used by the robot to understand the terrain configuration and then adapt to it.

Robots, however, are often required to operate in situations where wheels and legs are required. For robots used in Urban Search and Rescue (USAR), for example, operation environments include both flat ground such as concrete/wooden floors as well as rough terrains such as steps/stairs and unstructured surfaces such as rubble.



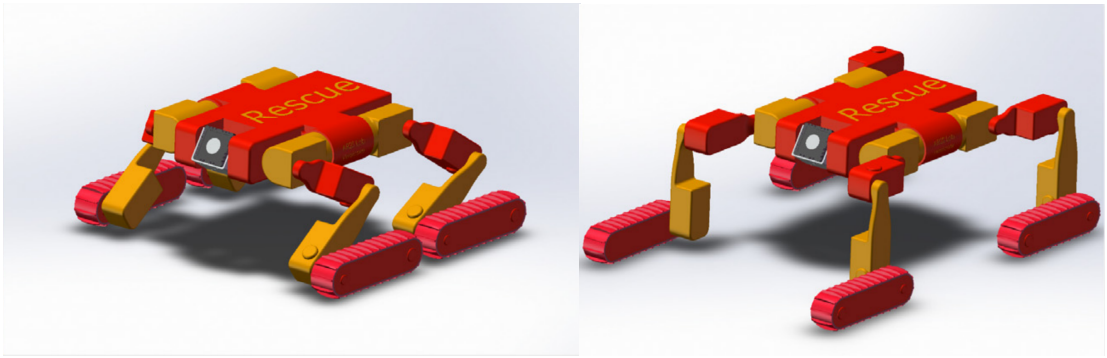
(a) Overcoming a difficult terrain (b) Overcoming a step

Figure 3.3: Example of obstacles

(a) Overcoming an unusual terrain (b) Overcoming a ladder

Figure 3.4: Example of obstacles

Legged robots offer good manoeuvrability in rough terrain but are inefficient on flat ground and need sophisticated control for any type of walking gait needed to be used. Wheeled/tracked robots are very efficient to move around on flat surfaces, but are inefficient and in some cases incapable of negotiating rough terrains. Hybrid solutions, combining the adaptability of legs with the efficiency of wheels/tracks, offer a fascinating compromise. Within the context of this thesis, robots having more than one locomotion mode are referred to as “Hybrid Robots”. Although not guaranteed,



(a) Track backward (b) Track forward

Figure 3.5: Track position examples

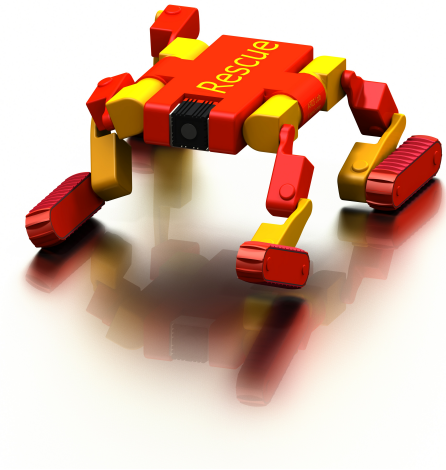
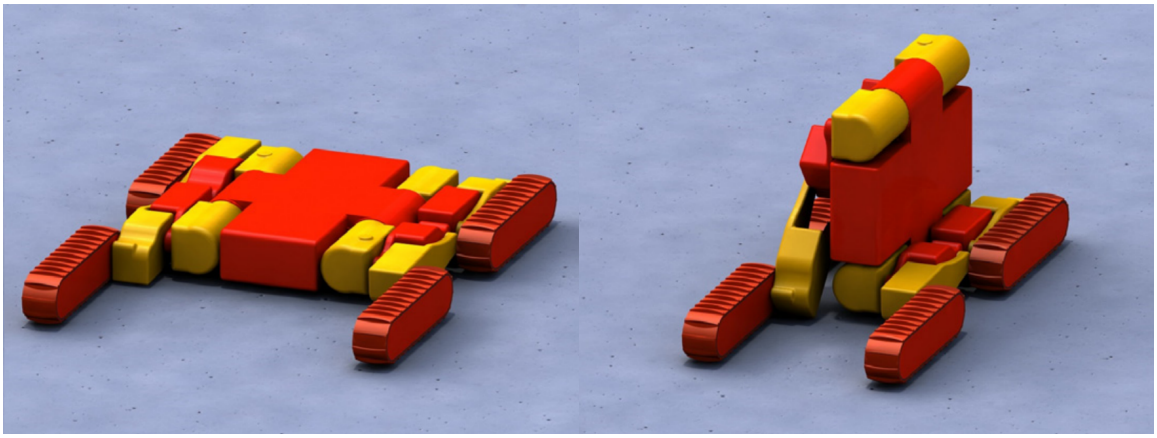


Figure 3.6: Changing configuration from rolling to walking

the use of hybrid mobile robots indicates a superiority over non-hybrid mobile robots for challenging tasks.



(a) Flat robot configuration (b) Robot configuration on a side

Figure 3.7: Particular configurations

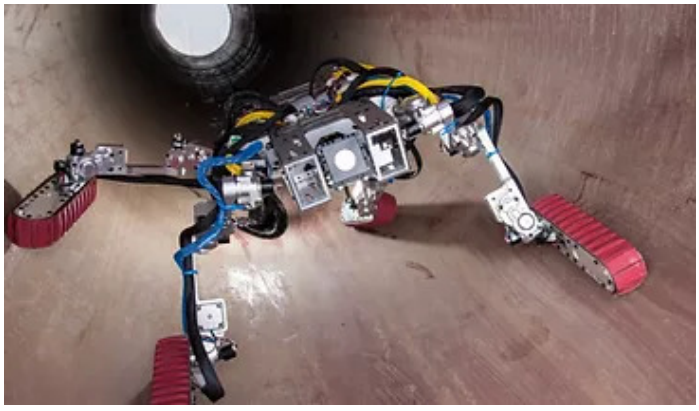


Figure 3.8: The Cricket robot going up a pipe.

Chapter 4

The Problem of Locomotion in Unstructured Terrains

This chapter describes the problems that an agent (in this case the Cricket robot) encounters in unstructured environments.

The need for the use of robotics in Urban Search and Rescue (USAR) and their deployment in confined spaces (e.g., mines, construction sites, collapsed buildings) is growing steadily. First USAR robots were adapted from robots developed to perform specific tasks such as duct inspection and bomb disposal that have proved useful for exploring voids within damaged infrastructure deemed too dangerous or too small for either canine or human entry.

Despite years of progress, the core design of robots currently in use for USAR purposes has deviated little, favouring perception/control development and optimization of

basic mechanical robotic systems to improve their performance instead of developing enhanced robotic tools capable of completing missions that current robots cannot. New robot designs, control, sensing, and navigation methodologies are needed to fully realize USAR robots and their deployment in fragile, complex, chaotic, and hazardous confined spaces.

The design of mobile robot locomotion systems for unstructured environments is often complicated, especially when they must move over uneven or soft terrains or climb barriers. Academic and business researchers have proposed several mechanical architectures for mobile robots. These solutions, of course, have diverse pros and cons. As a result, while designing a new mobile robot for a given application, a designer must analyse a wide range of viable technological options for its locomotion system, completing sophisticated and time-consuming analyses. The expected operating environments of a ground mobile robot must be analysed in the early design stages because they can fall into many different categories: indoor structured environments with the flat and compact ground, with or without stairs; outdoor environments with varying terrain firmness, with or without obstacles, and so on. As exposed in chapter 3, the robot used for this thesis aims to be as adaptable as possible to different kinds of terrains. The images and the degrees of freedom show the great adaptability of the robot.

The problem related to these robots with complex design, several joints and degrees of freedom, is that the manual control is complex and cumbersome. A human can not control them easily, and the risk of limiting the robot's potential is high. One possible solution is for the robot to learn on its own, how to move and disentangle itself within an unstructured environment.

4.1 Proposed Solution

The proposed solution describes a model-free approach, where the Cricket robot represent the agent in a discrete-time dynamical system. The robot is characterized by a state $s_t \in S$ at time t, where S is the set of possible states, takes in input the action $a_t \in A$, where A is the set of possible actions, and ends up at state s_{t+1} at time $t+1$. We assume that the system is controlled through a parameterized *policy* $\pi(a|s,t,\theta)$ that is followed for T steps (θ are the parameters of the policy). This document adopts episode-based, fixed horizon formulations. The solution considered uses a *deterministic policy;* this means that $\pi\left(a|s,t,\theta\right) \Rightarrow a = \pi(s,t|\theta)$. To evaluate the actions, we compute $r_{t+1} = r\left(s_t, a_t, s_{t+1}\right) \in \mathbb{R}$ which is called the *immediate reward* of state s_t at time t, taking the action a_t and reaching the state s_{t+1} at time $t+1$. To evaluate an entire episode, we consider the *long-term reward* (or *return*), which is defined as the sum of the immediate rewards:

$$\mathcal{R}_e = \sum_{t=0}^{T-1} r\left(s_t, a_t, s_{t+1}\right)$$

Where e is the episode. Moreover, the expected value of taking an action a in state s under the policy π is defined by the *Q-function*:

$$Q^{\pi}\left(s,a\right) = E_{\pi}[r_t|s_t = s, a_t = a] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\right]$$

Where $\gamma \in [0,1]$ is the *discount factor*. This means that when you get an immediate reward in the next step, it is discounted by a value γ. The more the discount factor is close to 1 the more the result considers the total reward, the more it is close to 0, the more it focuses on the immediate reward.

The proposed approach to train the agent is based on the DDPG algorithm (it is a type of algorithm in DRL that can be used to solve continuous action spaces problems,

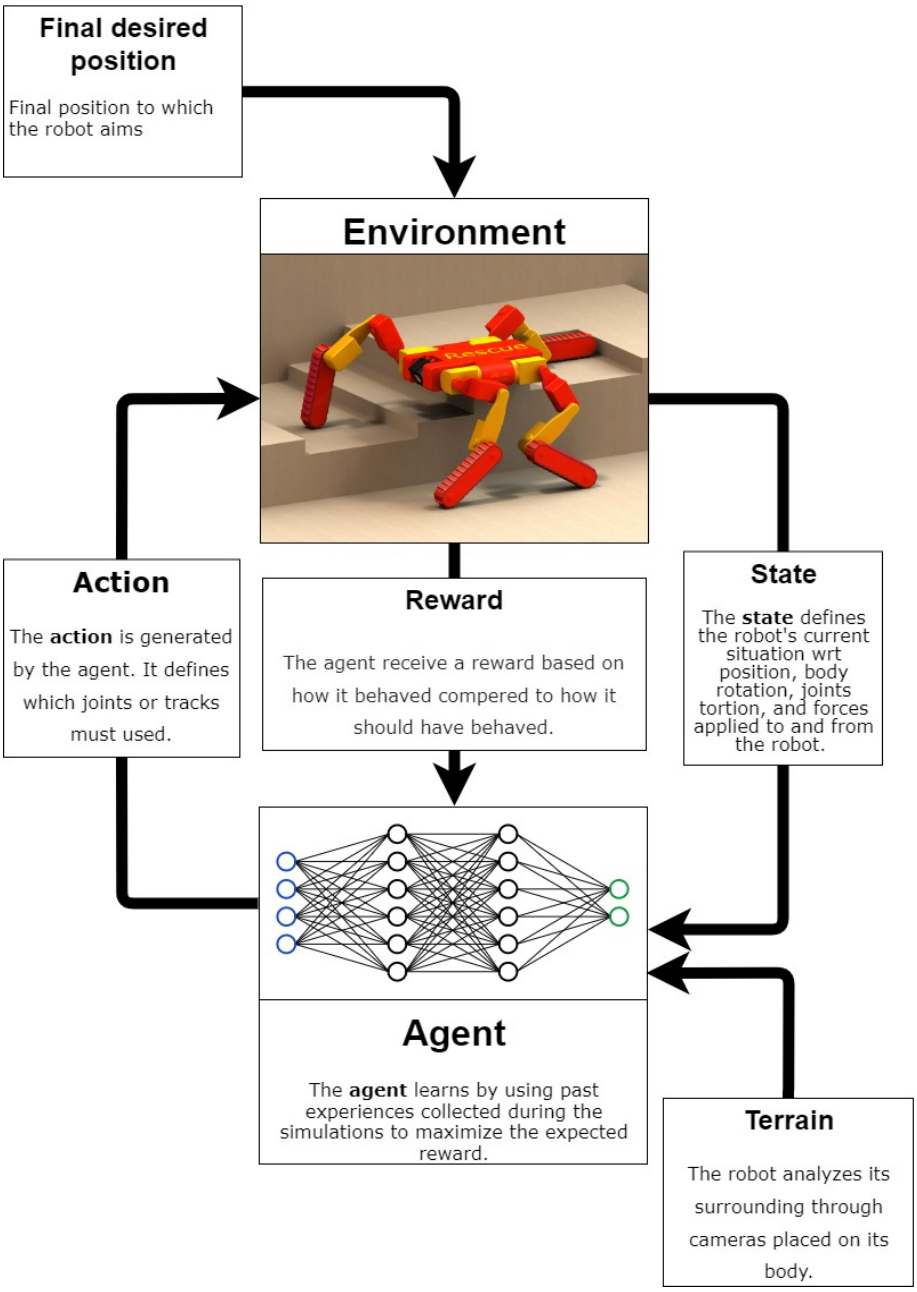


Figure 4.1: Overall solution's structure

see section 1.1). The purpose of being deterministic is to help the policy gradient to avoid random selection and to output a specific action value. Indeed, if you know the optimal action-value function $Q^*(s, a)$ then, in any given state, the optimal action can be defined as $a^*(s) = \ arg\ max_a Q^*(s, a)$. DDPG allows simultaneous learning of an approximator function for the optimal function $Q^*(s,\ a)$ and the optimal action $a^*(s)$. Considering a continuous action-space A, the DDPG algorithm sets up a gradient-based learning rule to achieve a policy $\pi(s)$ that allows approximating the optimization function as follows:

$$\max_a Q\left(s, a\right) \approx Q\left(s, \pi\left(s\right)\right)$$

$$\text{With } a \in A.$$

To develop the policy, the original DDPG algorithm does not consider the terrain on which the agent moves. The Cricket robot has two stereo-cameras, at the ends of the main body, which allow it to have a three-dimensional view of the world around it. This three-dimensional representation of the world is managed as a point-cloud and analysed through three-dimensional convolutional neural networks. In this way, the robot can autonomously understand the terrain composition and, eventually, recognize the obstacle and avoid collision with the environment. The management of the avoidance of obstacles, to avoid contact with them, is completely delegated to the reward function (explained in chapter 6) and, therefore, by the policy of the neural networks.

Another addition to the original DDPG algorithm is using an optimal final position, which is the "desired" position we want the robot to assume as a final goal. The reward function has been shaped to pursue this position and bound the robot to achieve it as faster and as efficiently as possible. At each step, the agent will receive a negative reward the further it will be from the final position; thus, considering the

angle spaced by joints, the position, in the euclidean space, the centre of mass, and the rotation, in terms of roll, pitch, and yaw.

In conclusion, the proposed solution aims to make the agent capable of computing a policy that allows it to reach a desired final position, regardless of the robot's initial position (tracks, joint angles, limb position, centre of mass rotation and location). The simulation is performed starting from a position where limbs and joints are positioned randomly, making, however, attention that these do not cause intersection with the environment or the robot itself. Each simulation episode ends when the robot stays motionless for a long time or the reward function's result reaches a certain threshold.

Figure 4.1 shows the overall structure of the solution. As can be seen, the structure follows the generic principle of reinforcement learning: an agent computes an action in the environment and receives a reward related to the action.

Chapter 5

Environment

In this section, the simulation and environment framework used to train the hybrid robot mentioned in section 3.1 are described.

The agent, explained below, interacts with the environment in discrete time steps. At each time t, the agent in state s_t chooses an action a_t which generates an effect in the environment. The environment is updated with the final position that the robot should seek, so, using this information, it can compute a reward r_{t+1} (through the reward function) and the agent moves to a new state s_{t+1}. The environment observes if the robot touches the terrain with limbs or body, or whether the robot makes movements that can lead to negative rewards. While the environment is known, an analytic solution is not defined, and the only way to collect information about the environment is to interact with it (in other words, the agent "learns" the environment). Therefore, the agent is forced to repeat the simulations until it develops a policy that allows it to reach an acceptable reward.

To implement the proposed reinforcement learning system, the environment must be defined. The environment is the world through which the agent moves and takes decisions. The environment uses the agent's current state s (or *observation*) and an action a as input to determine the agent's motion states. For this, a reward that defines the agent's next state s' is computed. The sequence of states defines the robot's movement. Inside the environment are encapsulated the simulation's physics rules and the laws that measure the effects of the actions taken by the robot through time. The evaluation of the robot is computed through the reward function (see chapter 6), which takes into account the final position of the robot, as well as other observations, and directs the search for a policy that optimizes the movement of the robot towards reaching the final position. The final desired position is expressed in terms of the level of joint bending (i.e., the angle of each 16 joints), and the position and orientation of the robot's centre of mass.

The training is performed through a deep reinforcement learning (DRL) system which combines artificial neural networks (ANN) with a reinforcement learning framework; refer to sections 6 and 7 from this thesis. This combined approach using ANNs and DRL unites approximation and target optimization, mapping states and actions to the rewards used by the robot to learn.

The experience gained by the robot is stored inside its "memory", and the robot can extract and exploit it. In this way, the robot will be able to extract and use better knowledge, concerning the environment. Hence, the proposed system needs a specific terrain and a desired final robot state to obtain a good policy that chooses the best action enabling the robot to adapt to the given terrain.

5.1 State, Action, Agent

The concept of **state** of the robot describes the current situation. It represents a finite number of elements which include:

- The position of the robot mass center $(X,\ Y,\ Z)$.
- The robot's body rotation, roll, pitch, and yaw (ψ, θ, ϕ).
- The 16 joint angle positions ($q_{h,j,i}$ where $h = \{left, right\},\ j = \{front, back\}$, $i = \{shoulder_1,\ shoulder_2,\ knee,\ ankle\}$).
- The robot's (center of mass) velocity $(v_x,\ x_y,\ v_z)$ and rotation angle rates $(\omega_\phi,\ \omega_\theta,\ \omega_\psi)$.
- The robot's track velocities $(\omega_\phi,\ \omega_\theta,\ \omega_\psi)$.
- The perpendicular forces generated by the contact of the 4 tracks on the terrain.

Let's focus on the normal forces applied to the robot's tracks as the robot touches the terrain. It is obvious that the force experienced by each track is not a single normal force on the whole track, but it is a force distributed along the track which depends on the kind of terrain with which the robot is in contact. For example, if the tracks lay on the sandy ground the distribution of normal forces is irregular, a similar reasoning can be carried out when the robot must deal with rocky or uneven terrain. In these cases, some parts of the track might measure different normal forces. To deal with this problem, we consider a fixed number of normal forces per track and then compute an approximation to, eventually, describe the robot state. Considering four normal forces per track we can define the following symbols $\overrightarrow{F}_{h,j,k}\ \ where\ \ h = \{left, right\},\ \ j = \{front, back\},\ \ k = \{1, 2, 3, 4\}$, which are a total of 16 normal

forces. In conclusion, the robot's state is represented by the 48 elements vector: $S = (X, Y, Z, \psi, \theta, \phi, q_{h,j,i},\ v_x,\ x_y,\ v_z,\ \omega_\phi,\ \omega_\theta,\ \omega_\psi,\ \overrightarrow{F}_{h,j,i})$.

The **action** generated by the agent's policy concerns the parts of the robot dedicated to movement. Hence, the agent's neural nets (policy) output is a vector of values that indicate how much the joints must rotate and how much the tracks must translate. We indicate these values with the symbols $\tau_{h,j,i}\ \ where\ h = \{left,\ \ right\},\ \ j = \{front,\ \ back\},\ \ i = \{shoulder_1,\ \ shoulder_2,\ \ knee,\ \ ankle\}$. Similarly, we refer to the rotation of the tracks with $\tau^T_{h,j}$. After several tests with the simulations, the value of these variables will become more and more accurate, and the system will understand that a minimum movement of a joint can drastically change the direction, roll, pitch, and yaw of the robot within the environment.

To effectively control the robot, we must define an **agent**. The agent evaluates the status to decide which action is better to take. Hence, the agent utilizes a policy π, that defines which action is better to take concerning the current state. On the other hand, the agent improves the policy during the training. This happens as a continuous feedback system with a reward (or penalty) at each time step. It is easy to understand that different designs of the reward function could lead to widely different results.

The agent learns by using past experiences collected during the simulations to maximize the expected reward. Indeed, while the simulation runs, the system will collect a set of states, actions, and rewards that it will use to train the policy inside the agent to further maximize the reward. The internal training of the agent is managed by the Actor-Critic networks that control the agent are explained in chapter 7.

Chapter 6

Reward Function

The following section shows the design of the reward function and an explanation of the mathematical formulations. Reward functions describe how the agent "ought" to behave. However, this function is a fundamental component of the system, and it can largely influence the behaviour of the robot. Therefore, good modelling needs particular attention and analysis of the problem. Changing the reward function's design may strongly affect the results. This function is necessary to prevent the robot from exposing itself to unnecessary or harmful movements, taking unwanted actions or spending too much time and energy to achieve its goal. The difficulty lies in balancing the generalization of the problem and being able to solve some predictable and avoidable behaviours. The reward function has been designed so that it collides with the surrounding environment and with the robot's other limbs or main body. Moreover, the function is modelled by considering the given information regarding the optimal final position: therefore, the bending of the joints, the centre of mass location (expressed in three-dimensional coordinates) and rotation (expressed in roll, pitch and yaw).

Type	**Mathematical Formulation**	**Description**
Self-collision penalty	$\alpha_i \left(q^i_{t-1}\right)^2$ $with\ \alpha \in \{0,1\}$	$\alpha_i = 1$ if the limbs connected to the joint i collide with some other robot's parts, 0 otherwise.
Environment-collision penalty	$\beta_i \left(q^i_{t-1}\right)^2$ $with\ \beta \in \{0,1\}$	$\beta_i = 1$ if the joint collides with the environment 0 otherwise.
Reward/penalty based on the direction of the last rotation	$k^i_t \left\lvert\tau^i_{t-1}\right\rvert$ $with\ k \in \{-1,1\}$	If the joint i at time t changes its rotation direction from $t-1$, then $k = -1$, 1 otherwise.
Difference with the optimal final joint torsion	$w^i_q \left(\Delta\ q^i_t\right)^2$	Evaluates the difference between the current level of twist of the robot joints and the desired ones. w_q is a constant weight.
Continued error	$w_\varepsilon \left(\sum_{i=0}^{x} \gamma^i \varepsilon_{t-i}\ \right)^2$	Where ε is the difference between $\hat{Q}_t(s,a)$ and $Q_t(s,a)$ at different timestamps. The function considers the last x steps. This error gives the robot a way to understand the error term and develop a policy accordingly. w_ϵ is a constant weight.
Time penalty	$w_t t$	The more time passes, the more the robot is penalized. w_t is a constant weight.

Type	**Mathematical Formulation**	**Description**
Movement toward the optimal final position	$w_{\mathcal{D}}(\mathcal{D}(P_f, P_t, P_{t-1}))^2$ where $\mathcal{D}(P_f, P_t, P_{t-1})$ is a function, P_f are the coordinate of the robot final position and P_t, P_{t-1} are the coordinate of the robot position at time t and $t-1$.	This factor gives a positive reward whether the center of mass at time t is closer to the final center of mass that the one at time $t-1$, negative otherwise. The closer it gets, the higher the reward, the farther it goes, the lower the reward. The coordinates are expressed as $x,\ y,\ z$, in a three-dimensional plane, while $w_{\mathcal{D}}$ represents a constant weight.
Difference with the optimal final rotation	$w_{\mathcal{T}}\left(\mathcal{T}(R_f, R_t, R_{t-1})\right)^2$ where $\mathcal{T}(R_t, R_f)$ is the function, R_f are the final robot rotation values (roll, pitch, yaw) and R_t, R_{t-1} are the robot rotation values at time t and $t-1$.	This factor gives a positive reward whether the robot's rotation values at time t are closer to the final robot's rotation values that the one at time $t-1$, negative otherwise. The closer it gets, the higher is the reward, the farther it goes, the lower is the reward. The rotation values roll, pitch and yaw are expressed as $\{\psi,\ \theta,\ \phi\}$ in a three-dimensional plane, while $w_{\mathcal{T}}$ represents a constant weight.

Table 6.1: Reward function summary

It is possible to breakdown the mathematical formulation reported in table 1. The **self-collision penalty** has the purpose of penalizing the robot when one or more limbs (or tracks) intersect each other (and therefore collide) or with the main body of the robot. Similar reasoning can be carried out for collisions with the environment (**Environment-collision penalty**): in case there is a limb (except for tracks) that collides with the ground or with parts against which the robot is not expected to collide, it is penalized. In the third line of the table, the **Reward/penalty based on the direction of the last rotation** can give either a positive or negative reward. The purpose of this reward/penalty is to punish the robot when it frequently changes the direction of joint rotation, and to incentive, it to maintain the current direction (in case this is wrong, other parameters come into play). Hence, the robot should not change the direction of its limbs too often and, instead, tries to get to the final position with the least number of movements, avoiding unnecessary direction changes or joint shaking. All penalty and reward calculations seen so far refer to a robot condition at $t-1$. Although, the operation $w_q^i \left(\Delta\ q_t^i\right)^2$, reported in row 4 of the table, considers the robot state at time t and evaluates how far each joint is from the desired twist for the optimal robot's state (i.e., the final robot's state). These penalties incentive the robot to identify the final position and make the appropriate movements to reach it. On the other hand, the **continued error** operation $w_\varepsilon \left(\sum_{i=0}^{x} \gamma^i \varepsilon_{t-i}\ \right)^2$ considers the loss calculated in the last x time steps; this error gives the robot a way to understand the error term and develop a policy accordingly. The value of ϵ is computed with the difference between the estimated value $\hat{Q}_t(s,a)$ from the critical network, and the effective $Q_t(s,a)$ measured at the end of the simulation at time t. The value of x would not have to be exaggeratedly great, to avoid useless or harmful calculations to the aim of the learning. This calculation considers values back in the time; this allows the robot to better understand the error term and develop a policy accordingly. The **time penalty** incentives the robot to reach the final status as fast as it can. If an exit

condition is set during the robot training, this type of penalty could be problematic, as it could lead the robot to reach this condition rather than the desired end state. For example, let's assume the condition "*if* the robot stands still for 5 seconds, *then* the simulation ends" is considered as a *simulation end condition*. The robot may prefer to remain motionless rather than make movements that would result in lengthening the duration of the simulation, and thus its total penalty. To overcome this problem, it is necessary to regulate with cure the value constant value of w_t. Moreover, it is necessary to introduce incentives for movement: the last two rows of table 6.1 consider the position and rotation of the robot's centre of mass relative to the desired position and rotation. The position is calculated in a three-dimensional plane (x, y, z), while the rotation considers roll, pitch, and yaw $(\psi,\ \theta,\ \phi)$. Indeed, this function members give a reward in case of movement that pursues the goal of approaching and being as similar as possible to the final desired position. All weight constants, denoted by w, must be calibrated and balanced to achieve a satisfactory result.

With this information, it is possible to proceed with the mathematical formulation of the complete reward function:

For simplicity, let's start by considering the difference between two points in a three-dimensional space:

$$dist(P_i, P_j) = \sqrt{(P_j^x - P_i^x)^2 + (P_j^y - P_i^y)^2 + (P_j^z - P_i^z)^2}$$

Where $P_{\{i,j\}}$ is a generic point in a three-dimensional space.

The goal is to design a function's member that gives a positive reward when the robot (and particularly its centre of mass) moves towards the final position and negative otherwise. To do so, we can write the following function:

$$\mathcal{D}(P_f, P_t, P_{t-1}) = dist(P_{t-1}, P_f) - dist(P_t, P_f)$$

Where P_t and P_{t-1} is the position of the robot center of mass position at time t and $t-1$; while P_f is the desired robot center of mass. Considering that we want to emphasize this member of the equation we can square it, and then multiply it with a constant weight to adjust its influence within the equation. Eventually, it is possible to modify the $dist(P_i, P_j)$ function and add different weights for each Cartesian direction, to give more importance to a direction rather than another one.

Furthermore, a 3D body can rotate on the three orthogonal axes; so we continue with a similar evaluation regarding roll, pitch, and yaw:

- A roll is a counterclockwise rotation of α about the x-axis. The rotation matrix is given by

$$R^{\psi} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{pmatrix}$$

- A pitch is a counterclockwise rotation of β about the y-axis. The rotation matrix is given by

$$R^{\theta} = \begin{pmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{pmatrix}$$

- A yaw is a counterclockwise rotation of γ about the z-axis. The rotation matrix is given by

$$R^{\phi} = \begin{pmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

As for the position we can define the rotation reword function member as:

$$\mathcal{T}(R_f, R_t, R_{t-1}) = rot\_diff(R_{t-1}, R_f) - rot\_diff(R_t, R_f)$$

Where $rot\_diff(R_{\{t,t-1\}}, R_f)$ measure the difference between the robot rotation at time t or $t-1$ and the final optimal rotation. The difference between the two rotation is computed in terms of roll, pitch, and yaw, respectively $\Delta\psi, \Delta\theta, \Delta\phi$. Again, this function member is weighted (with $w_{\mathcal{T}}$) to adjust its influence in the final function. Furthermore, just as with the $dist(P_i, P_j)$ function, it is possible to modify the $rot\_diff(R_i, R_j)$ function to give different priorities to the various directions of rotation.

Finally, we can complete the reward function with all other operations. The final reward function is showed below:

$$\mathcal{R}_t = -\sum_{i=0}^{no\_track} [\alpha_i \left(q_{t-1}^i\right)^2 + \beta_i \left(q_{t-1}^i\right)^2 + \kappa_i \left|\tau_{t-1}^i\right| + w_q^i \left(\Delta q_t^i\right)^2] +$$
$$-w_\varepsilon \left(\sum_{i=0}^{x} \gamma^i \varepsilon_{t-i}\right)^2 - w_t t + w_{\mathcal{D}} \mathcal{D}(P_f, P_t, P_{t-1})^2 + w_{\mathcal{T}} \mathcal{T}(R_f, R_t, R_{t-1})^2$$

It is possible to see that the reward function is severe (we can almost call it a penalty function). With this function, the robot will hardly get a reward greater than zero, but this is not a problem considering that the robot will try to minimize the penalty in any case by reaching the lowest punishment possible.

Chapter 7

Actor and Critic

An introduction of Actor-Critic approach has been done in section 1.1, as subsection of "Fundamentals of Reinforcement Learning".
Control policies based on Deep Neural Networks (DNNs) show successfully applications in both autonomous aerial vehicles [41] and swimming control motion. Given the recent developments of reinforcement learning applied to control policy, it is possible to divide the algorithms into three macro-areas:

Direct policy approximation: DNNs can be used to approximate a control policy $\pi(s)$ directly. For example, from different initial states, trajectory optimization can be used to compute families of optima control solutions, with each trajectory yielding a large number of data points suitable for supervised learning.

The second macro area is the *Value based* method. In this case, the system tries to approximate the optimal value function (the mapping between an action and its value). One example of this method is the Q-learning algorithm [42]. This solution is popular for decision problems and its discrete action spaces. Indeed a DNN can be used to learn a complete state-action value function by approximating a set of value

functions, one for each specific action. This approach has become famous after the publication of Mnih et al [43]. The capacity of an AI to learn to play a vast suite of Atari games at a human level of expertise, using only raw screen images and score input is a remarkable achievement.

Finally, the *Policy gradient* approach uses DNNs to represent both the Q-function $Q(s, a)$, and the policy $\pi(s)$ in scenarios of continuous actions learned in the absence of an oracle. This results in a single composite network $Q(s, \pi(s))$, which allows value-gradients to be backpropagated to the control policy and do provide a method for policy improvement. This approach is also applied to stochastic policy methods that can include both model-free and model-based methods [44]. Other proposals show the use of policy-gradient DNN-RL for simulated robot soccer [45].

The *Actor-Critic* (AC), as a Temporal Difference version of the Policy gradient, aims to take advantage of all the positive aspects of the approaches listed before. It consists of two main DNNs: Actor and Critic. The actor behaves as a policy network, that takes the agent's state and the terrain as input, and the evaluation network is called a Critic network. The Actor network selects continuous actions as a result of the policy gradient (DDPG explained in chapter 4.1), and the Critic network assesses whether and how much the consequences of the action are positive or negative by calculating the value function. So, the actor acts as an arg max over the possible *action-values* (or *Q-values*) of all actions, which makes the algorithm "deterministic". Indeed, the actor-network can be also called "deterministic policy network" [46].

$$\mathcal{A}(s, e) = argmax_a Q(s, a)$$

where A represents the actor network, s the state, and e the terrain.

To promote exploration, some Gaussian noise is added to the action determined by the policy. The actor output is fed into the critic network to compute the estimated

Q-value i.e., $\hat{Q}(s, a|e)$. Hence, the critic takes in input the state, the terrain, and the last action of the agent. The estimated Q-value computed represents the value generated by an action taken in a particular robot's state in the given terrain. In literature, this kind of network is sometimes called "Q-network" [47].

$$\mathcal{C}\,(s, e, a) = \hat{Q}(s, a|e)$$

Where C is the critic network, s the state, and e the terrain.

The Actor-network and the Critic network are two separate networks that share the state information. During the simulation, it is possible to observe the real Q-value measured from the simulator. Performing trial and error, we can observe and collect the actual measured values $Q(s, a)$, then compare these results with the estimations ($\hat{Q}(s, a)$) computed by the critic network. Finally, the loss function (or error measurement) is computed, and its result is used to back-propagate and train the networks. The last computation is explained in section 7.2. It is crucial to evaluate the action strategy in the Critic network, which is more conducive to the convergence and stability of the current Actor-network. The above features ensure that the AC

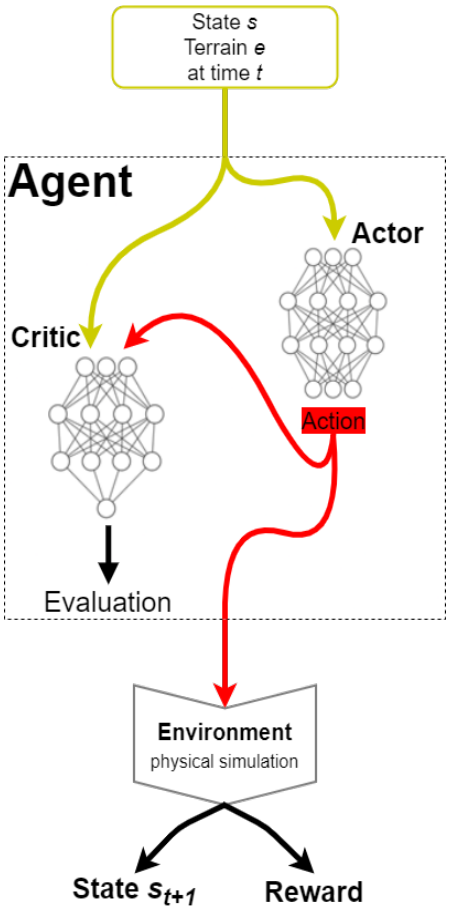


Figure 7.1: Actor-critic networks structure.

algorithm can obtain the optimal action strategy with the gradient estimation at a lower variance.

This technique uses an off-policy approach to train a deterministic policy. Given this, the agent would not probably not try a wide variety of actions to obtain useful learning signals if it were to explore on-policy at first. To improve the exploration, a solution is to artificially introduce noise to the policies' actions during the training. As suggested in [44], a mean-zero Gaussian noise works perfectly for this purpose. To facilitate getting higher-quality training data, it may be possible to reduce the scale of the noise over the course of training. This generates a new action from the continuous action space:

$$a = \mathcal{A}(s, e) + \mathcal{N}(0, S^2)$$

Where S are pre-specified scales for each action parameter.

7.1 Target Networks

The network update equations are interdependent on the values computed by the neural network, making it prone to diverge. Therefore, two *target networks*, for both critic and actor, stabilize the learning process. The original networks are used to output the actions in real-time, evaluate actions, and update network parameters through training, which includes the original Actor network and original Critic network, respectively. The target networks include the target Actor network and target Critic network, which are used to update the value network system and the Actor-network system, but not to carry out training and updating of network parameters. Hence, the algorithm uses a total of four neural networks, a Critic network $\mathcal{C}$, a target Critic network $\mathcal{C}'$, an Actor-network $\mathcal{A}$, and a target Actor network $\mathcal{A}'$. These target networks have the same structure and initialization parameters as the respective original

networks. Moreover, these networks work as time-delayed networks compared to the main networks and use '*soft-updates*' (explained in section 7.2) to perform training that follows the original networks. When the algorithm starts, it initializes the main neural network with random weights and these target networks as a copy of the main networks.

7.2 Algorithm Discussion

The DDPG algorithm takes the information of the initial state as the input, and the output result is the action policy strategy calculated by the algorithm. Then, the random noise is added to the action generated by the policy to obtain the final output action. This process improves the exploration and avoids the overfitting of neural networks. When starting the task, the agent outputs an action according to the current state s_t. The reward function is designed (see section 6), and the action is evaluated. The more an action is beneficial, the more the penalty is minimized. Afterwards, all the simulations and results computed by the network are stored in a **reply buffer**. This data structure is fundamental to sample past experiences and update the parameters of the neural networks. At each iteration, it stores a transition tuple of the form (s_t, a_t, r_t, s_{t+1}) where s is the state, a the action, r the reward, and t the timestamp. This reply buffer works as a finite-sized array that can return mini-batches of experience with which the network is updated. At the same time, the neural networks continuously adjust policy by randomly extracting mini-batches of data from the replay buffer and using a gradient descent approach to update and iterate network parameters. The *Algorithm 1* shows the pseudocode that implements the main algorithm.

DDPG combines with DQN on the premise of the AC algorithm to further enhance the stability and effectiveness of network training, which makes it more conducive

in the field of solving the problem of continuous state and action space. Meanwhile, DDPG subdivides the network structure into the original network and target network.

The **neural network update** happens as follows: after computing the *Bellman Equations* (line 17 of the pseudocode) [11], the *next-state Q-values* are computed with the target critic (value) and actor (policy) network. Then it minimizes the MSE between the updated Q-value and the original Q-value, the latter computed by the critic network $\mathcal{C}$.

Regarding the **policy loss update**, we aim to maximize the expected return (as in the code we use *s,* *a*, and *e* respectively as *state*, *action*, and *terrain*):

$$J(\theta) = \mathbb{E}\left[Q\left(s, \mathcal{A}\left(s, e\right)\right)\right]$$

To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter. Applying the chain rule, we obtain:

$$\nabla_{\theta^{\mathcal{A}}} J(\theta) \approx \nabla_a \, \mathrm{C}\left(s, a, e\right) \nabla_{\theta^{\mathcal{A}}} \mathcal{A}\left(s, e \middle| \theta^{\mathcal{A}}\right)$$

Finally, we can take the mean of the sum of the gradients computed from the mini-batches, and we obtain the update equation in line 20.

$$\nabla_{\theta^{\mathcal{A}}} J \approx \frac{1}{n} \sum_i \nabla_a \mathcal{C}\left(s, \mathcal{A}\left(s, e \middle| \theta^{\mathcal{A}}\right), e \middle| \theta^{\mathcal{C}}\right) \nabla_{\theta^{\mathcal{A}}} \mathcal{A}\left(s, e \middle| \theta^{\mathcal{A}}\right)$$

As the last step, the algorithm performs the **soft update** for the target neural networks. This process consists of copying the parameters from the original networks with slight modifications. This update is computed once per main network update by *Polyak averaging*:

$$\theta^{\text{target}} \leftarrow \tau \theta^{\text{original}} + (1 - \tau)\, \theta^{\text{target}}$$

With the approximation coefficient $0 \leq \tau \leq 1$, usually close to 0.

Where θ^{target} and θ^{original} represent respectively the parameters of the target network (actor or critic), and the original network.

Figure 7.2 shows a diagram that explicates the behaviour brought back in the pseudocode (the symbology in the picture is the same one used in the pseudocode).

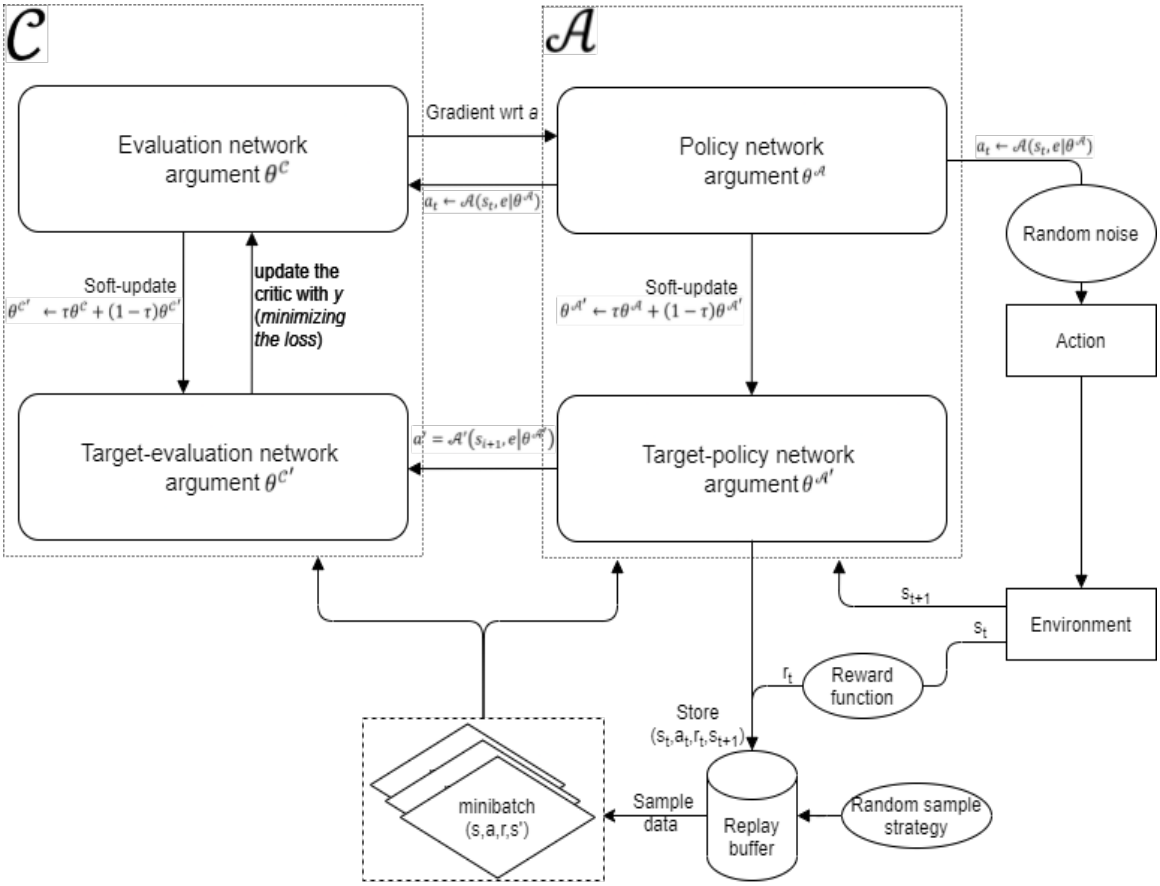


Figure 7.2: Algorithm shown graphically.

Algorithm 1 DDPG Pseudocode

Initialize actor and critic networks $\mathcal{A}(s, e|\theta^{\mathcal{A}})$ and $\mathcal{C}(s, a, e|\theta^{\mathcal{C}})$ with weights $\theta^{\mathcal{A}}$ and $\theta^{\mathcal{C}}$
Initialize target network $\mathcal{A}'$ and $\mathcal{C}'$ with random weights $\theta^{\mathcal{A}'} \leftarrow \theta^{\mathcal{A}}$ and $\theta^{\mathcal{C}'} \leftarrow \theta^{\mathcal{C}}$
Initialize replay buffer $\mathcal{R}$
for episodes in $N_{episodes}$ **do**
    Procedure RobotInit
    Generate a random stable starting state s on the unstructured environment
    End procedure RobotInit

    Procedure Simulation
    for $t = 0$ to T **do** ▷ we define $a = action,\ s = state,\ r = reward,\ e = terrain$
        Generate action $a_t \leftarrow \mathcal{A}(s_t, e|\theta^{\mathcal{A}})$
        Execute action $a_t \Rightarrow$ observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
        if *updateCondition* is True **then**
            Sample random **minibatch** of n transition (s, a, r, s') from $\mathcal{R}$
            for (s_i, a_i, r_i, s_{i+1}) in minibatch **do**
                set $y_i = r_i + \gamma \cdot \mathcal{C}'(s_{i+1}, \mathcal{A}'(s_{i+1}, e|\theta^{\mathcal{A}'}), e|\theta^{\mathcal{C}'})$
            end for
            Update **critic** by minimizing the loss:

$$L = \frac{1}{n} \sum_i (y_i - \mathcal{C}(s_i, a_i, e|\theta^{\mathcal{C}}))^2$$

            Update the **actor** policy using the sampled policy gradient:

$$\nabla_{\theta^{\mathcal{A}}} J \approx \frac{1}{n} \sum_i \nabla_a \mathcal{C}(s, a, e|\theta^{\mathcal{C}})|_{s=s_i, a=\mathcal{A}(s)} \nabla_{\theta^{\mathcal{A}}} \mathcal{A}(s, e|\theta^{\mathcal{A}})|_{s_i}$$

            Update **target networks**: ▷ Considering $0 \leq \tau \leq 1$

$$\theta^{\mathcal{A}'} \leftarrow \tau\theta^{\mathcal{A}} + (1 - \tau)\theta^{\mathcal{A}'}$$

$$\theta^{\mathcal{C}'} \leftarrow \tau\theta^{\mathcal{C}} + (1 - \tau)\theta^{\mathcal{C}'}$$

        end if
        checkStopConditions()
    end for
    end procedure Simulation
end for

7.3 Neural Networks Structures

We already stated that the policy is an actor neural network and along with it is a critic neural network and each of these networks has numerous hidden layers. We need several neurons to mimic the high dimensional function, which is necessary to map all the observations and the terrain to the action of this nonlinear environment. On the other hand, using an excessive number of neurons means spending more time training the expensive logic. In addition, the architecture of the network is significant in functional complexity. It is necessary to study and perform several tests to find the perfect balance between the number of hidden layers that makes training possible and efficient.

Figure 7.1 shows an extremely simplified general structure of the actor-critic model. Both networks take as input the observation of the environment. The analysis of the robot state happens through dense layers (see section 5.1). The Cricket robot has two stereo-cameras, at the ends of the main body, which allow it to have a three-dimensional view of the world around it. This three-dimensional representation of the world is managed as a point-cloud and analysed through three-dimensional convolutional neural networks (CNN). In this way, the robot can autonomously understand the terrain composition and, eventually, recognize the obstacle and avoid collision with the environment. The output of the 3D CNN is flattened and then added to the output of the DNN that takes in the input of the robot state. The terrain reading can be carried out at each time step or arbitrary intervals. The Actor output corresponds to the width and speed of the robot's joints and tracks, for 40 output nodes. The output produced by the Actor, i.e., the robot action, is used as input for the critic network. Therefore, the critic's network takes as input the robot state, the terrain features, and the action computed by the actor. In addition, its network has another "branch" that takes as input the action generated by the actor. All these branches

merge, and the critic's final output is, therefore, a single value.

Description	**Symbols**	**Number of values**
Robot's center of mass position	X, Y, Z	3
Robot's velocity	v_x, v_y, v_z	3
Robot's center of mass rotation angle	$\phi,\ \theta, \psi$	3
Robot's center of mass rotation angle rate	$\omega_\phi,\ \omega_\theta,\ \omega_\psi$	3
legs encoders	$q_{R_{\text{ji}}},\ q_{L_{\text{ji}}} \vert i \in [0,1] \wedge j \in\ [0,4]$ R=right side, L=left side	16
Track velocities	$\omega_{R_i}, \omega_{L_i} \vert\ i \in [0,1]$	4
4 contact forces per track	$\overrightarrow{F}_{R_{i,j}},\ \overrightarrow{F}_{R_{i,j}}$ R=right side, L=left side	16
Errors computed (see section 6)*	$\varepsilon_t\ with\ t \in [0,4]$	5
Total values:	-	53

Table 7.1: Variables in inputs for the neural networks (without the terrain).
* This is not considered as part of the robot's state.

7.3.1 Actor's Structure

The structure of the actor consists of two different branches of the network. One analyzes the state of the robot. Hence, it is a series of fully connected layers interspersed with ReLU functions. The other branch analyzes the terrain features using convolutional layers. Since the robot moves in three-dimensional confined spaces, the

convolutions are also three-dimensional. The last layers of the two branches are concatenated through a *merge layer* and the result (the action to perform) is computed with a *hyperbolic function*. The structure of this network is shown in figure 7.3. The terrain input layer's dimension depends on the type of terrain (as well as the kernel for the 3D convolutional network), while the input layer that analyzes the robot state is always 53 (see section 7.3).

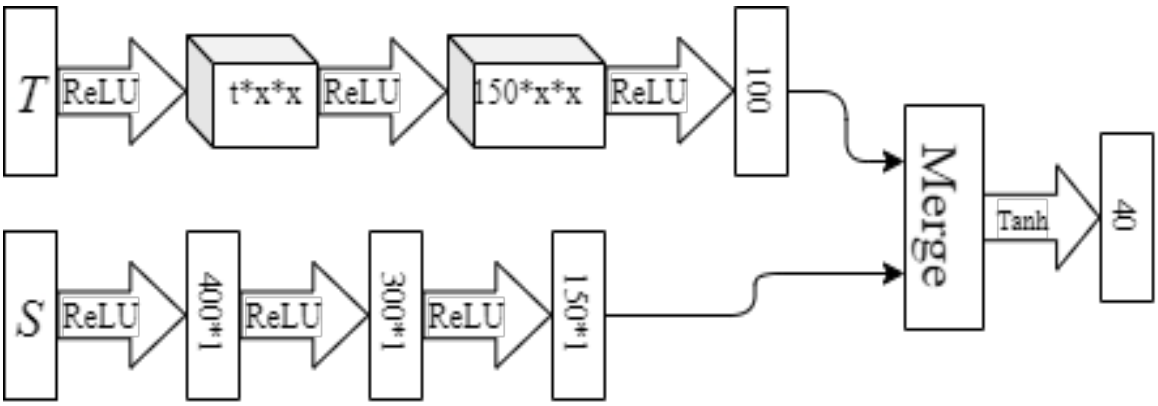


Figure 7.3: Schematic illustration of the actor deep neural network. T and S are the input terrain and character state. The output represents the angles spaced by the joints and the spacing speed.

7.3.2 Critic's Structure

The structure of the critic is similar to the Actor's, however, there are two major differences. Firstly, the critic has one more branch than the actor. This is necessary to elaborate on the action generated by the actor. Secondly, the activation function on the output layer is a ReLU, in this way we obtain just one value for the output (which is the expected value for the simulation). The structure of this network is shown in figure 7.4. The terrain input layer's dimension depends on the type of terrain (as well as the kernel for the 3D convolutional network), while the input layer that analyzes the robot state is always 53 (see section 7.3). The input layer for the

action is 40.

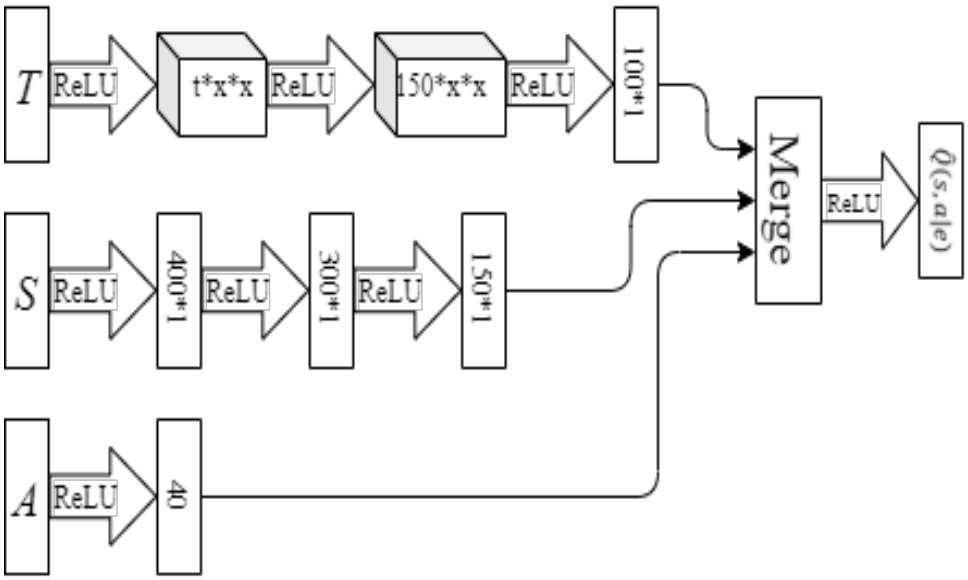


Figure 7.4: Schematic illustration of the critic deep neural network. T and S are the input terrain and character state and A is the action generated by the actor network. The output is one value and represents the estimated value of the action.

Chapter 8

Implementation

As explained in the previous chapters the algorithm implemented is the classical DDPG with the addition of Convolutional NNs to perform terrain features extraction for the agent, and specifies the final state (so not just the location) that defines the optimal position, joint angles and orientation characteristics of the robot, which affects the evaluation performed in the environment. The main challenge resided in the modelling of the state and reward function. As well as the study and the implementation of the 3d physics environments for the simulations and the three-dimensional design of the robot.

Summarizing the general characteristic of the deep reinforcement learning system (such as the dimension of the state and the actor and critic networks etc.). The state is composed of 48 elements (section 5.1), and the reward function uses some elements of the state (such as position and rotation, joint angles) and other observations (continued error, environment collision, time, final desired position), the actor-network takes in input the state and the terrain (represented as a three-dimensional cloud-

point $400 \times 400 \times 400$), while the critic takes in input the same as the actor-network plus the latter's output. The desired final pose is represented as a set of angles (one for each joint of the robots), and values for the position and orientation of the centre of mass (expressed in $x,\ y,\ z$ and $\psi,\ \theta,\ \phi$ respectively).

The algorithm takes into input the terrain (as a point-cloud) and the desired pose of the robot, while the robot is placed on the terrain in a quasi-random pose. "Quasi-"random because the pose is generated respecting the physics, therefore, avoiding interpenetration of the robot's limbs with the terrain or the main body. The algorithm starts with a *warm-up* phase, where the agent performs random actions while filling the memory buffer and the policy (i.e. the actor, but the critic too) doesn't get updated. After this phase, the backtracking is performed based on the batch size.

The actor and critic buffers are initialized with 50k tuples from a random policy that selects an action uniformly from the initial action set for each cycle. Each of the initial actions is manually associated with a subset of the available actors using the actor bias. When recording an initial experience tuple, μ will be randomly assigned to be one of the actors in its respective set.

This work has been carried out with the use of OpenAI [48] for robotics, using ROS (Robotic Operating System[1]), Gazebo[2], and PyBullet[3]. A whitepaper about ROS and Gazebo simulations for robotics is available at [14]. A comparison between these two physic simulators for robotics is available at [49].

[1] www.ros.org
[2] www.gazebosim.org
[3] www.pybullet.org

8.1 URDF and Xacro Model

The Unified Robot Description Format (URDF) is an XML file format used in ROS to describe all elements of a robot. URDF was used to describe the robot so that it could be used in a simulation using ROS with Gazebo or other physical engines. Two different models were developed for this project, the first one is a simple geometric 3D model, that consists in parallelepipeds, cylinders, and cubes, which make up the robot's limbs and joints. The second model, in collaboration with HEBI Robotics, shows a complex and precise modelling, which is optimal for accurate results, but is cumbersome and heavy during simulations.

While URDFs are a useful and standardized format in ROS, they lack many features and have not been updated to deal with the evolving needs of robotics. URDF can only specify the kinematic and dynamic properties of a single robot in isolation. URDF can not specify the pose of the robot itself within a world. It is also not a universal description format since it cannot specify joint loops (parallel linkages), and it lacks friction and other properties. Additionally, it cannot specify things that are not robots, such as lights, heightmaps, etc. On the implementation side, the URDF syntax breaks proper formatting with heavy use of XML attributes, which in turn makes URDF more inflexible. There is also no mechanism for backward compatibility. Xacro (XML Macros) Xacro is an XML macro language. With Xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions.

8.1.1 Training Model

This model consists of a Xacro file that is compiled and generates a simple model of the robot, composed of geometric

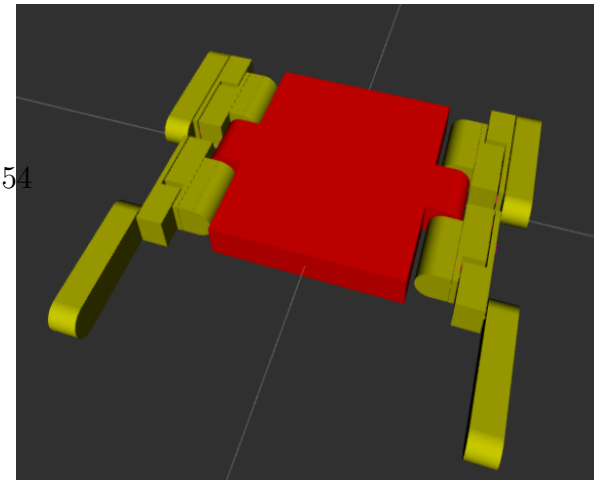


Figure 8.1: Actor-critic networks struc-

figures, with approximate weights and characteristics. It does not have textures and does not indicate the materials of which the various parts of the robot are composed, consequently, unbalanced static and dynamic friction forces are also considered during the simulation. This simple model is useful during code testing and debugging, as its modelling does not overly burden the graphics processors and they can devote themselves to memory allocation for the neural networks.

8.1.2 Collaboration with HEBI robotics

The second model was created in collaboration with HEBI Robotics and consists of a model reliable and reflects reality. As mentioned above, this model is computationally heavier to manage and move. It is recommended to use this model once the code has been tested and it is working. This model is shown in figure 8.2, it is possible to compare the CAD model to a picture of the real robot showed in figure 8.3.

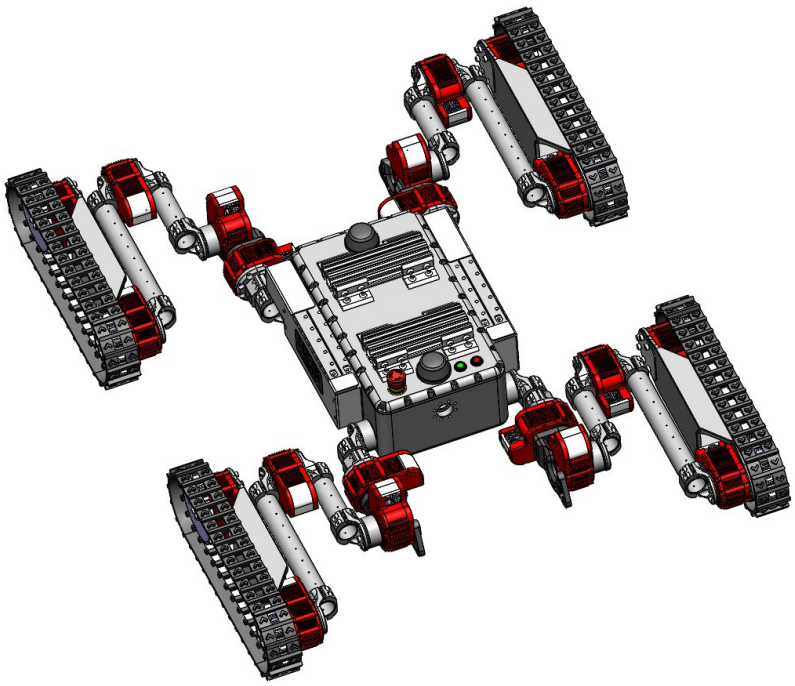


Figure 8.2: Final CAD model in collaboration with HEBI Robotics.

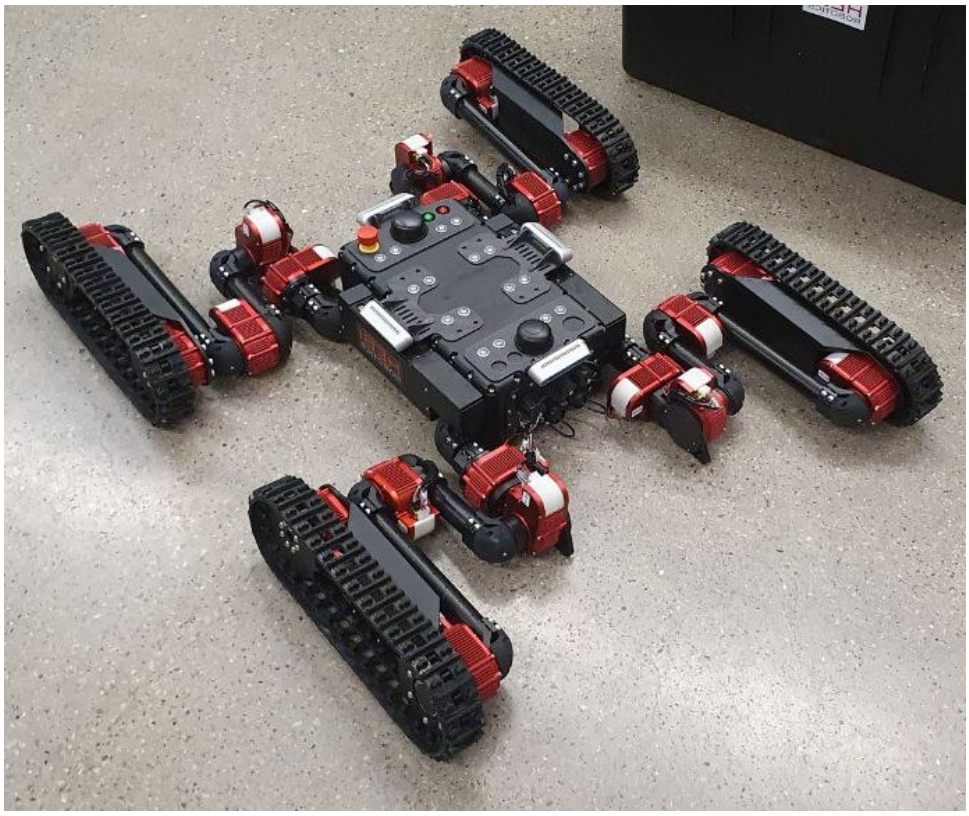


Figure 8.3: Real version of the CAD model.

8.2 Reinforcement Learning Gym

Two core concepts are the agent and the environment. OpenAI only provides an abstraction for the environment, not for the agent. This choice allows the developer to implement different styles of the agent interface. Indeed, on OpenAI it is possible to implement online learning as well as offline learning algorithms, where, for example, the agent takes some *observations* as input and at each timestep performs learning updates incrementally instead of a batch update (as it is described in this document). Using this framework, any changes to the environment will increase the version number for the algorithms that are being tested. Otherwise, it wouldn't be possible to

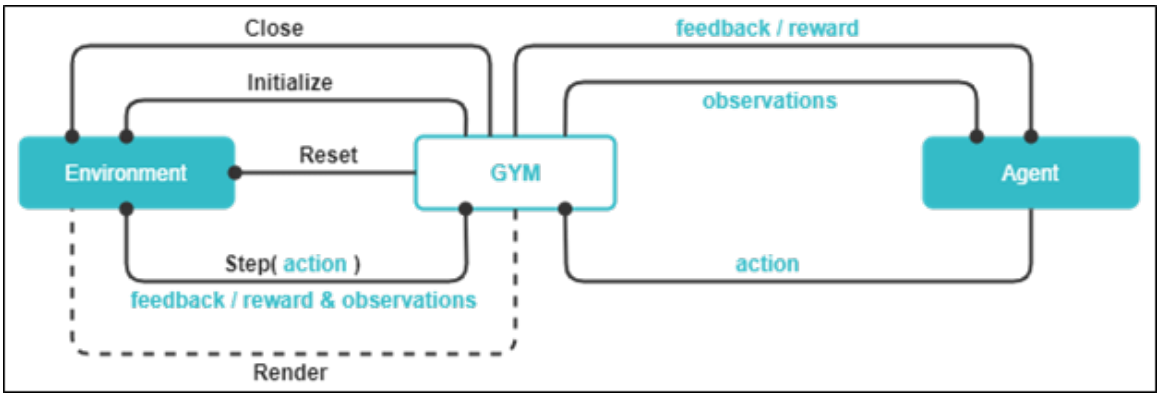


Figure 8.4: Structure of the reinforcement learning gym that communicates with the agent and the environment.

compare the results obtained in a different environment. Moreover, considering that OpenAI manages what concerns the logic of the environment, it is possible to implement an algorithm on this framework using a certain physical simulation engine (such as Gazebo) and easily change it with another one (e.g., pyBullet), without rewriting the logic of the algorithm.

The performance of an RL algorithm on an environment can be measured along two axes: first, the final performance; second, the amount of time it takes to learn—the sample complexity. To be more specific, final performance refers to the average reward per episode, after learning is complete. Learning time can be measured in multiple ways, one simple scheme is to count the number of episodes before a threshold level of average performance is exceeded. This threshold is chosen per environment in an ad-hoc way, for example, as 90% of the maximum performance achievable by a very heavily trained agent. Both final performance and sample complexity are very interesting, however, arbitrary amounts of computation can be used to boost final performance, making it a comparison of computational resources rather than algorithm quality. Figure 8.4 explains the overall structure of the simulation gym and how this interacts with the agent and the environment.

8.3 Hyper-parameter Tuning

As commonly happens for reinforcement learning algorithms, experiments need many episodes, and this algorithm is no exception. Each episode divides into hundreds or thousands of steps, which is when the agent generates action and the environment evaluates it (so, a step in the physical simulation is performed). If it turns out that the robot is standing still, the algorithm stops the episode and punishes the agent with a negative reward.

Each episode lasts at most 2000 steps/action, and it can end with either failure (i.e. the robot stand still) or hitting the limit of the step.

The weights used to regulate the reward function have been tuned by observing the robot's actions within the simulations and trying to adjust the function regarding the terrain to face. The remaining hyper-parameters values adopted came from [50].

8.4 Practical Tests and Results

To design and create the terrains I used "Blender"[4]. This software allows for the creation of 3D models and exports them in different formats. Since we need to represent the terrain as a cloud point, Blender allows us to export the terrain as a set of points in 3D coordinates. Furthermore, it can export the terrain mash as a *urdf* file, which can be easily represented in the simulation.

Running a simulation consists of running the code by passing as input the path to the folder containing the terrain files and the file containing the final robot pose information.

[4]https://www.blender.org/

The simulations performed and the corresponding results, expressed in terms of reward, are listed below. All the plots 8.5, 8.6, and 8.7 shoe the average reward and the stand deviation for each episode, and the global trend of the reward. In the following plots, the value of the rewards (y axis) is expressed in yE+6, while the number of episodes is expressed in xE+2, where y and x are the reward values and the episodes respectively.

- **Flat terrain:**

  The first and most basic experiment consists of starting from a random position and developing a policy to get up and remain stable in an almost-flat terrain. See picture 8.5a.

  As the picture 8.8a shows, the robot quickly achieved a steady and relatively high reward.

- **Sloping terrain** (front and side):

  In this test, the robot tries to stand on a 30° sloping terrain in both front and side position (see pictures 8.6a) and 8.6b).

  Again, the solutions in 8.9a and 8.9b the agent can develop a policy that allows it to reach a good reward level quickly and maintain it over episodes. The difference between front and side does not seem to plague the effectiveness of the policy very much.

- **Step terrain** (front and side):

  In picture 8.7 the robot aims to overcome a step as high as its legs. The experiment has been performed in both frontward and side-ward directions.

  In this case, the results 8.10 show that the reward cannot rise up and often suffers severe readjustments. It is likely that the robot's contact with the ground

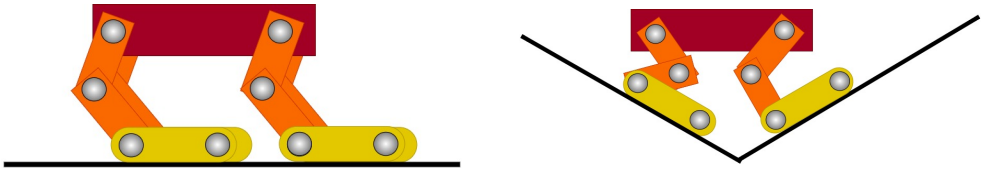
is more frequent and leads to punishments from the reward function. Indeed, the reward value is much lower than those seen so far.

- **V-shaped terrain:**

  This terrain is similar to the "slope terrain", but with two sides, shaped like a "V" (image 8.5b).

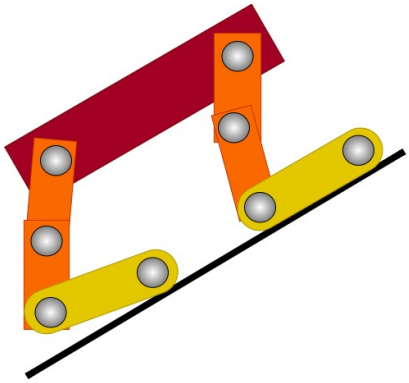
  Although, this terrain may look like a complex terrain allows the agent to achieve a similar level of reward as obtained for flat terrain. The graph 8.8b can be indeed compared with 8.8a.

A single simulation can take up to a few days. Therefore, there was no time to develop and test other terrains. Despite this, some terrains were planned for future experiments: a terrain composed of cubes (inspired by the video game Q*bert), one with a higher slope (45°), and another similar to the one in the image 3.4a.

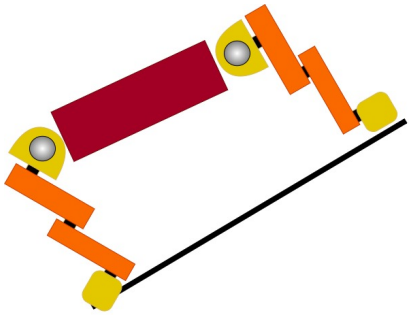


(a) Cricket standing on a flat terrain. (b) Cricket standing on a V-shaped terrain.

Figure 8.5: Example of Cricket's final desired configuration for a given terrain

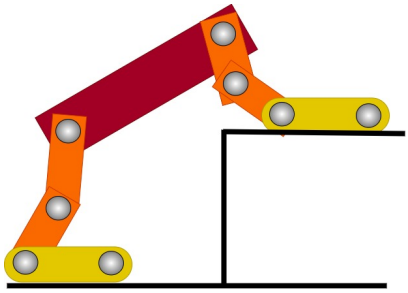


(a) Cricket standing on a 30° sloped terrain.

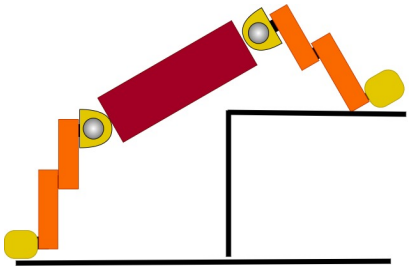


(b) Cricket standing on a 30° sloped terrain on one side.

Figure 8.6: Example of Cricket's final desired configuration for a given terrain

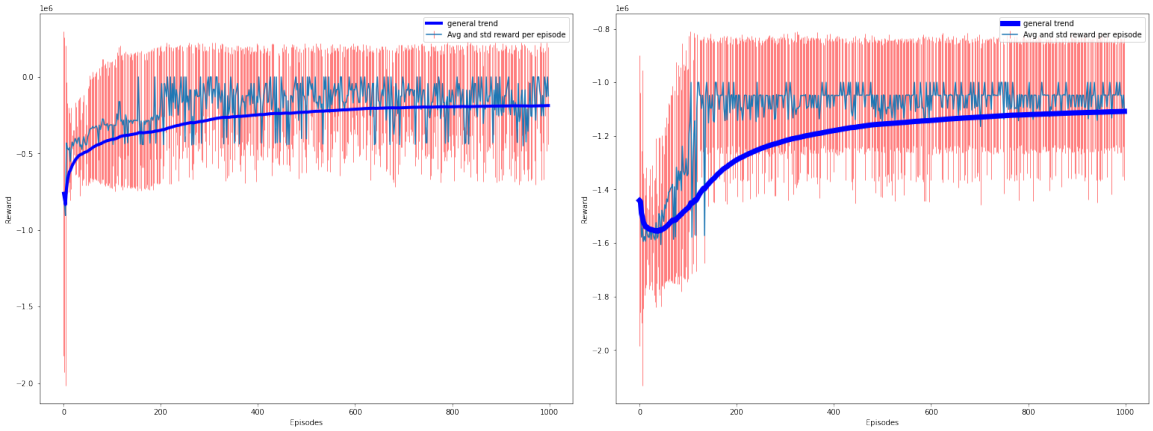


(a) Cricket overcoming a step.



(b) Cricket overcoming a step on the side.

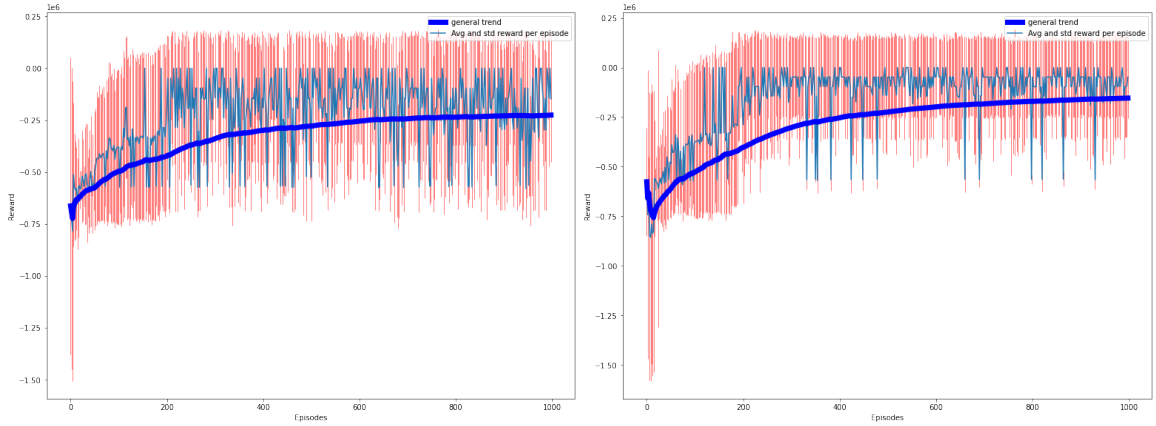
Figure 8.7: Example of Cricket's final desired configuration for a given terrain



(a) Reward trend for flat terrain.

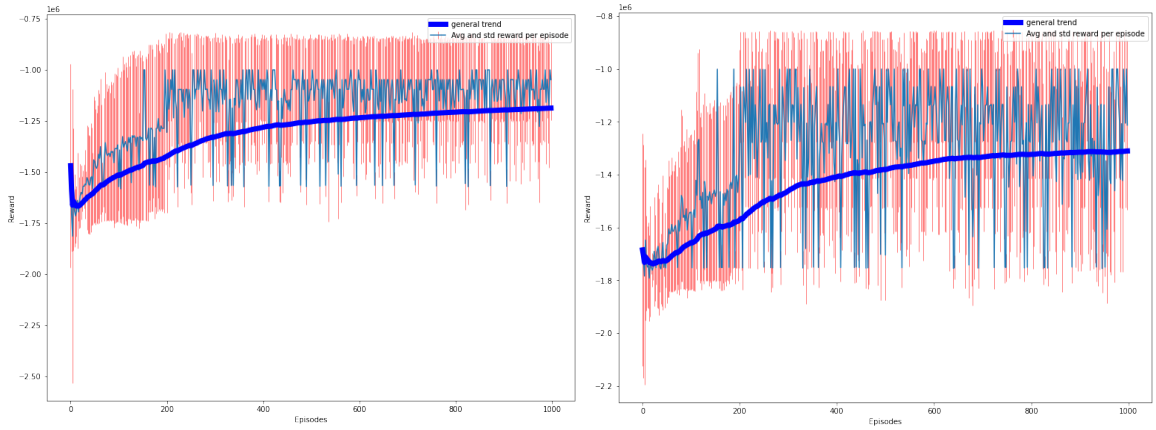
(b) Reward trend for V-shaped terrain.

Figure 8.8: Reward trend for the specified terrains. The episodes are expressed are in form E+2.



(a) Reward trend for slope terrain on front. (b) Reward trend for slope terrain on a side.

Figure 8.9: Reward trend for the specified terrains. The episodes are expressed are in form E+2.



(a) Reward trend to reach the position to overcome a step on front.

(b) Reward trend to reach the position to overcome a step on a side.

Figure 8.10: Reward trend for the specified terrains. The episodes are expressed are in form E+2.

Chapter 9

Conclusion

This thesis is inspired by the well-known deep deterministic policy gradient algorithm and modifies it to create a framework for computing a policy in a specific terrain. This framework allows an agent to achieve an optimal pose previously defined. Moreover, it introduces how this algorithm integrates into a hierarchical learning system that allows the robot to locomote in unstructured environments (which is the lab's goal). Indeed, whether the robot must move in an environment where the terrain is known but complex (e.g., on the moon, or the interior of a mine), it is possible to prepare test plots and have the robot learn how to perform the appropriate movements to adapt to them on its own. As mentioned above, this thesis enriches the DDPG algorithm by adding terrain analysis using sensors placed on the robot and neural networks. In addition, the final position of the robot is a key component not used by DDPG. The unique reward function, studied specifically for the Cricket robot, demonstrates its effectiveness, as the robot is not driven to stand still. The key challenge was modelling the state and reward function. As well as the research and development of 3D physics environments for simulations and the three-dimensional design of the

robot.

As explained in the introduction, this thesis is part of a larger project carried out by the UVS laboratory. The work done has paved the way for future researchers and students who would like to approach solving the problem of autonomous locomotion of hybrid robots in unstructured environments. The ability to adapt to different environments with complete autonomy could affect the industry as ordinary humans lives. The architecture enables the development of control strategies for extremely dynamic terrain adaptive locomotion that can operate directly with high-dimensional state descriptions. This eliminates the requirement to construct small hand-crafted feature descriptors and allows policies to be learned for classes of terrain where developing compact feature descriptors may be difficult.

9.1 Future Works

There are still numerous limitations to be addressed. The policy needs a human to model the final optimal position; this process could be assisted by locomotion algorithms. Moreover, the approach needs to be applied to several kinds of terrains before becoming useful, otherwise, the robot can learn how to adapt to specific terrain before starting a mission. The learning process itself is also quite time-consuming, often requiring several hours or to adapt to a single environment and is performed independently for each policy. Although the same reward is used across all motions, this is still currently based on a manually defined metric. It would be interesting to see the characteristic of the policy deployed on the real system and compare them to the one computed in the simulations. The transposition of the algorithm into the real-world version (i.e., not simulated) will require special attention and, probably,

an update of the algorithm, since external agents, such as wind, weather, temperature, or even other physical aspects such as friction or hardware overheating, are not taken into account. Control policies for complicated terrains may need to be learned progressively via curriculum learning, especially for scenarios where the initial random policy performs so poorly that no meaningful directions for improvement can be found. Self-paced learning is also a promising direction, where the terrain difficulty is increased once a desired level of competence is achieved with the current terrain difficulty. It may be possible to design the terrain generator to work in concert with the learning process by synthesizing terrains with a bias towards situations that are known to be problematic. Other paths toward more data-efficient learning include the ability to transfer aspects of learning solutions between classes of terrain, developing an explicit reduced-dimensionality action space, and learning models of the dynamics. It would also be interesting to explore the co-evolution of the character and its control policies. It is not yet clear how to best enable control of the motion style. The parameterization of the action space, the initial bootstrap actions, and the reward function all provide some influence over the final motion styles of the control policy. Available reference motions could be used to help develop the initial actions or used to help design style rewards. This problem can be addressed using **genetic algorithm**, in literature, various examples show the capabilities of this approach to optimize the rewards and the joint movements. An interesting project would be to create principled ways for merging several controllers, each trained for a specific type of terrain. This would enable the successful establishment of control strategies based on divide-and-conquer tactics.

Having stayed in touch with the lab, I know that a new student, building on the work presented in this thesis, will use a mixture of actor-critic experts (MACE) approach that learns terrain-adaptive dynamic locomotion skills using high-dimensional state and terrain descriptions as input. As showed in [50].

Appendix A

Computer components

The experiments and training have been carried out on a PC with the following components:

- CPU: i5-10500 Processor, 3.1GHz w/ 6 Cores / 12 Threads
- GPU: NVIDIA Quadro RTX 4000 8GB - CUDA Parallel-Processing Cores 2,304
- Motherboard: ASUS PRIME B460M-A
- SO: Ubuntu 18.04 (Bionic Beaver)

Appendix B

Code structure

Although the robot was modelled with CAD and URDF, the algorithm, environment and model management of the robot were developed in python. To implement the characteristics of the deep neural networks, the algorithm uses Pytorch accelerated with CUDA. The code is divided as follows:

- **ddpg.py** contains the core of the main algorithm.
- **main.py** contains the logic that connects the environment (CricketEnv) to the algorithm (DDPG).
- **gym_cricket/** this folder contains the parts of the code that communicate with the simulation environment.
    - **assets/** this folder contains the *terrains*, the *urdf models*, and the classes that describe the robot itself and its characteristics inside the simulation.
        - **cricket_abs.py** contains an abstract class that encapsulates the main function used by the simulation to obtain information about the robot.

It contains the getters and the setters for the joint positions, the normal forces, the collisions information, and so on.

- ◇ **cricket.py** implements the abstract class ***cricket_abs.*** This class loads the training URDF file (section 8.1.1). Hence, it uses a simplified version of the cricket robot. Light and simple to run and edit. Useful for test and debug.
- ◇ **hebi_cricket.py** implements the abstract class ***cricket_abs.*** This class implements and loads the URDF file of the complete version of the robot (section 8.1.2). This class has been used to realize the final tests.
- ◇ **cricketGoal.py** and **hebi_cricketGoal.py** extend respectively the cricket and hebi_cricket classes described above. They contain the final desired position for the cricket robot. Used by the reward function to give the robot a direction to follow.
- ◇ **terrains/** contains the folders for the different kinds of terrain.
- ◇ **urdf/** contains the *.urdf* and *.xacro* files for the simplified cricket robot, and the folder that incorporates the final robot CAD and URDF description.

- ○ **envs/**
  - ◇ **cricket_env.py** contains the logic of the environment. Hence, the reward function, the logic behind the steps in the simulation, and control the state of the cricket robot.

- **neural_networks/** see the ddpg algorithm for the actor-critic network
  - ○ **actor_nn.py** Describes the actor network
  - ○ **critic_nn.py** Describes the critic network

- **utils/** this folder contains all the auxiliary classes and functions (as the buffer for the ddpg or function used by CUDA).

In the UML diagrams in figures 22 and 23, the two main sets of classes used by the algorithm are shown. Both the CricketEnv class and the DDPG class are initialized and used by the main file. If one wanted to modify the simulation system one would only need to modify the CricketEnv file. Given the complexity and function variables common to the TrainCricket and HebiCricket classes, it was decided to develop an abstract class that would implement the common variables and logic to make the code maintainable and streamlined. As for the two classes described above, the conversion from XML file to array takes place within the main and is passed as input to the classes that use it.

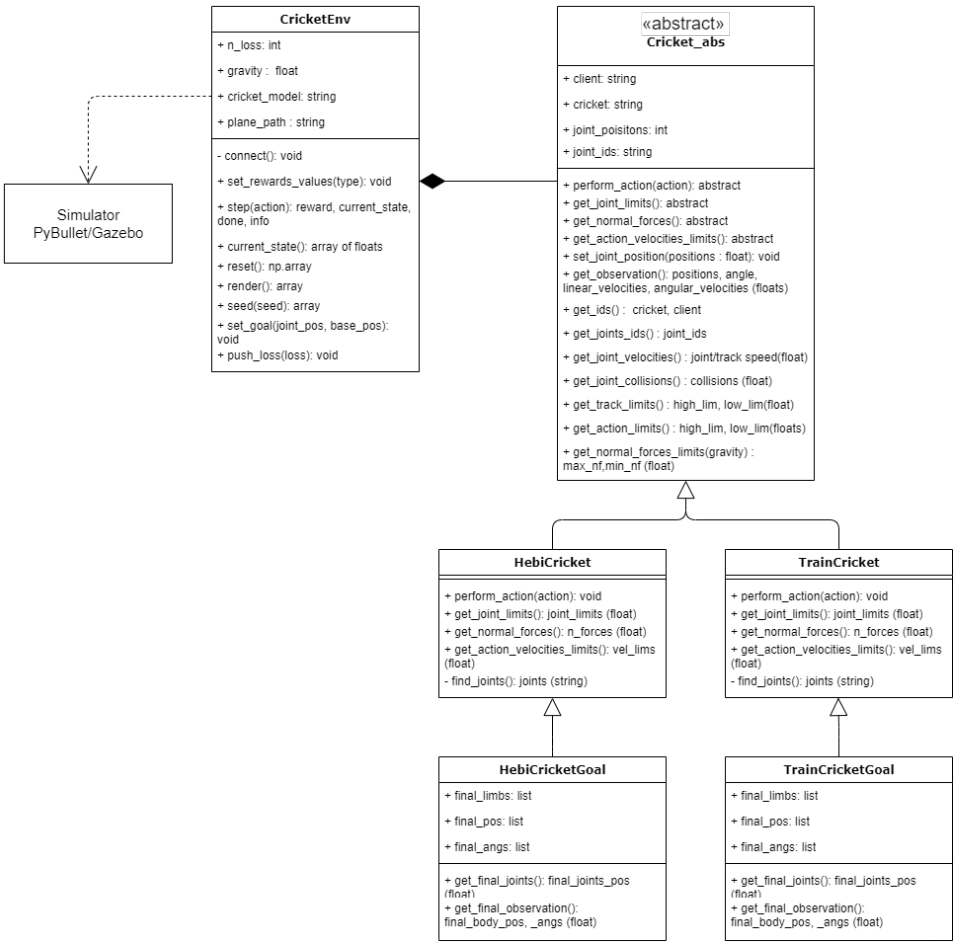


Figure B.1: UML representation of the Cricket environment that encapsulates the information about the robot state and controls the joints to perform actions.

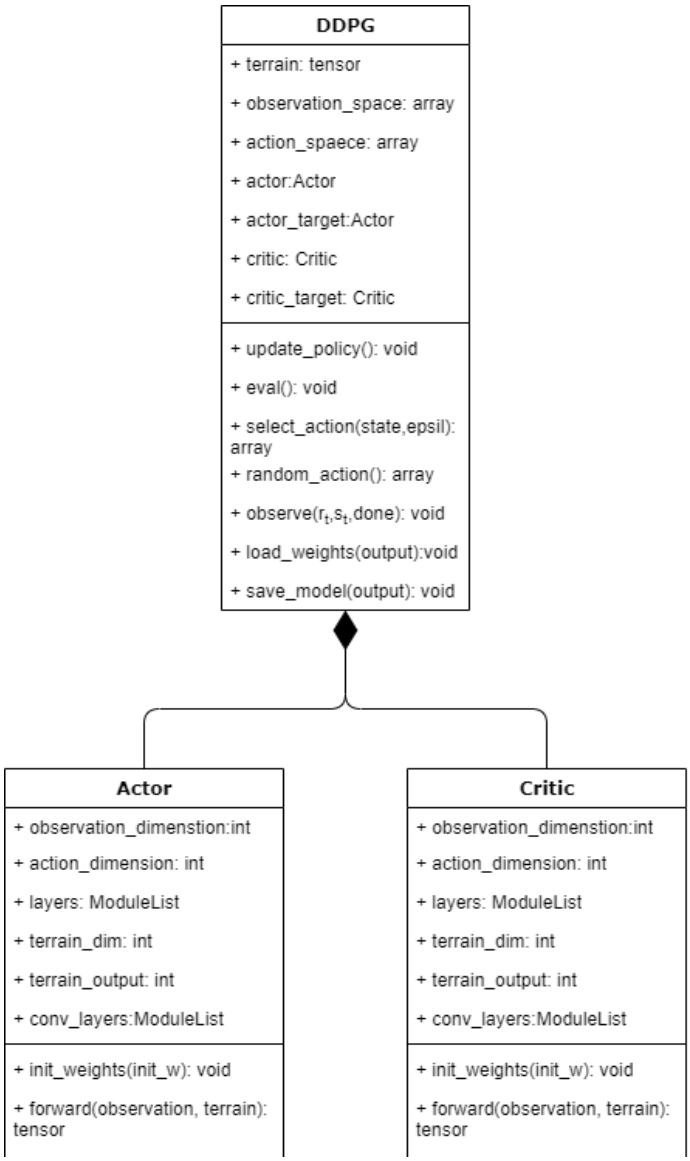


Figure B.2: UML representation of the DDPG algorithm and the function to perform updates and predictions of the neural networks.

B.1 Technologies used

This section lists the technologies, the tools and the languages used to realize the code, the environment and the virtual representation of the robot.

As said in section 8, the development of the 3D models of the robots has been done within the **Robotic Operating System** environment. The 3D model of the robot dedicated to the training of the neural network has been realized following the procedure of creation of XML models with the use of macros made available by the package Xacro of the Robotic Operative System [1]. The repository encapsulates some bash scripts to run the Gazebo simulations and the virtual room (RVIZ [2]) where it is possible to manually control the limbs and tracks of the robot; hence, movements and positions.

The model developed in collaboration with HEBI robotics uses ROS packages made by HEBI Robotics itself. To run the model it is necessary to install the API made available by the company (the repository comprehends a ReadMe that explains how to do so). It is necessary to have **C++** installed in the system to launch the simulation environment. Modules programmed using **MATLAB** and **Simulink** ensure and verify every aspect of the simulation, from perception to motion. The final three-dimensional design of the robot has been realized with **CAD**.

The neural networks and the deep reinforcement learning system, together with the virtual gym environment, have been developed with **Python 3.7**, using the following (non-standard) libraries:

- gym (openAI)
- pybullet
- numpy
- torch (Pytorch)

[1] http://wiki.ros.org/xacro
[2] http://wiki.ros.org/rviz

- pywavefront (to read specific type of data)
- matplotlib
- argparse
- setuptools
- pathlib

finally, the training of the neural networks has been carried out thanks to the use of CUDA version 11.

The code is realized in a way that from the command line it is possible to determine the ground that is wanted to use, the dimension and depth of the neural nets that compose the system and to personalize the destinations for the saving or loading of the nets.

The soils on which the simulations are performed are modelled in 3D using "Blender" software [3]. The modeled terrain is exported in three formats: Object (.obj), ".urdf", and *Wavefront Material Template Library* (.mtl). The first two formats are used by the physical simulation engines (PyBullet and Gazebo (ROS)) as the basic ground for their simulations. The ".mtl" file is imported into the deep learning system as a point cloud and then read by the 3D convolutional neural network.

[3] https://www.blender.org/

Appendix C

Glossary

Reinforcement learning and Artificial Intelligence can be defined by several concepts. The following section summarizes and briefly explains the meaning of the main terms used in this document.

- **Agent**: the entity that uses a policy to maximize the expected return gained from transitioning between states of the environment.
- **Action**: the mechanism by which the agent transitions between states of the environment. The agent chooses the action by using a policy.
- **Environment**: In reinforcement learning, the world contains the agent and allows the agent to observe that world's state. For example, the represented world can be a game like chess or a physical world like an unstructured terrain. When the agent applies an action to the environment, then the environment transitions between states.
- **State (or *observation*)**: the parameter values that describe the current configuration of the environment, which the agent uses to choose an action.

- **Reward**: the numerical result of taking an action in a state, as defined by the environment.
- **Policy**: an agent's probabilistic mapping from states to actions.
- **State-action value function (or *Q-function*)**: the function that predicts the expected return from taking an action in a state and then following a given policy.
- **Action-value (or *Q-value*)**: Following a policy π the action-value function returns the value, i.e., the expected return for using action a in a certain state s.
- **Discount factor**: The discount factor affects how much weight it gives to future rewards in the value function. A discount factor $\gamma = 0$ will result in state-action values representing the immediate reward, while a higher discount factor $\gamma = 0.9$ will result in the values representing the cumulative discounted future reward an agent expects to receive (under a given policy π).

Bibliography

[1] Matteo Sostero. Automation and robots in services: Review of data and taxonomy. JRC Working Papers Series on Labour, Education and Technology 2020/14, Seville, 2020.

[2] Stephan Weyer, Mathias Schmitt, Moritz Ohmer, and Dominic Gorecky. Towards industry 4.0 - standardization as the crucial challenge for highly modular, multi-vendor production systems. volume 48, pages 579–584, 12 2015.

[3] R. X. Gao C. Xu J. F. Arinez, Q. Chang and J. Zhang. Artificial intelligence in advanced manufacturing: Current status and future outlook. 2020.

[4] Vasso Marinoudi, Claus G. Sørensen, Simon Pearson, and Dionysis Bochtis. Robotics and labour in agriculture. a context consideration. *Biosystems Engineering*, 184:111–121, 2019.

[5] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

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

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

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[9] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In *ICML*, 2016.

[10] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1, 06 2014.

[11] Hans Josef Pesch and Roland Bulirsch. The maximum principle, bellman's equation and caratheodory's work. *Journal of Optimization Theory and Applications*, 80:203–229, 02 1994.

[12] Ivaylo Popov, Nicolas Manfred Otto Heess, Timothy P. Lillicrap, Roland Hafner, Gabriel Barth-Maron, Matej Vecerík, Thomas Lampe, Yuval Tassa, Tom Erez, and Martin A. Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *ArXiv*, abs/1704.03073, 2017.

[13] Huanming Zhang, Zhengyong Jiang, and Jionglong Su. A deep deterministic policy gradient-based strategy for stocks portfolio management, 2021.

[14] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo, 2017.

[15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[17] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[18] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. Crawling in rogue's dungeons with (partitioned) a3c. In *Machine Learning, Optimization, and Data Science*, pages 264–275. Springer International Publishing, 2019.

[19] Aviral Kumar, Anikait Singh, Stephen Tian, Chelsea Finn, and Sergey Levine. A workflow for offline model-free robotic reinforcement learning, 2021.

[20] Ryan Lawhead and Abhijit Gosavi. A bounded actor-critic reinforcement learning algorithm applied to airline revenue management. *Engineering Applications of Artificial Intelligence*, 06 2019.

[21] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[22] Baolin Peng, Xiujun Li, Jianfeng Gao, Jingjing Liu, Yun-Nung Chen, and Kam-Fai Wong. Adversarial advantage actor-critic model for task-completion dialogue policy learning. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6149–6153. IEEE, 2018.

[23] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[24] Xiaoteng Ma, Li Xia, Zhengyuan Zhou, Jun Yang, and Qianchuan Zhao. Dsac: Distributional soft actor critic for risk-sensitive reinforcement learning, 2020.

[25] Konstantinos Chatzilygeroudis, Vassilis Vassiliades, Freek Stulp, Sylvain Calinon, and Jean-Baptiste Mouret. A survey on policy search algorithms for learning robot controllers in a handful of trials, 2019.

[26] L. Bruzzone and G. Quaglia. Review article: locomotion systems for ground mobile robots in unstructured environments. *Mechanical Sciences*, 3(2):49–62, 2012.

[27] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[28] S Lim and Jason Teo. Recent advances on locomotion mechanisms of hybrid mobile robots. *WSEAS Transactions on systems*, 14:11–25, 2015.

[29] Chenghui Nie, Xavier Pacheco Corcho, and Matthew Spenko. Robots on the move: Versatility and complexity in mobile robot locomotion. *IEEE Robotics & Automation Magazine*, 20(4):72–82, 2013.

[30] Ludovic Daler, Stefano Mintchev, Cesare Stefanini, and Dario Floreano. A bioinspired multi-modal flying and walking robot. *Bioinspiration & biomimetics*, 10(1):016005, 2015.

[31] Arash Kalantari and Matthew Spenko. Modeling and performance assessment of the hytaq, a hybrid terrestrial/aerial quadrotor. *IEEE Transactions on Robotics*, 30(5):1278–1285, 2014.

[32] Auke Jan Ijspeert, Alessandro Crespi, Dimitri Ryczko, and Jean-Marie Cabelguen. From swimming to walking with a salamander robot driven by a spinal cord model. *science*, 315(5817):1416–1420, 2007.

[33] Masashi Takahashi, Kan Yoneda, and Shigeo Hirose. Rough terrain locomotion of a leg-wheel hybrid quadruped robot. In *Proceedings 2006 IEEE International*

Conference on Robotics and Automation, 2006. ICRA 2006., pages 1090–1095. IEEE, 2006.

[34] Mark Yim, Ying Zhang, and David Duff. Modular robots. *IEEE Spectrum*, 39(2):30–34, 2002.

[35] Arthur W Mahoney, Patrick L Anderson, Philip J Swaney, Fabien Maldonado, and Robert J Webster. Reconfigurable parallel continuum robots for incisionless surgery. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4330–4336. IEEE, 2016.

[36] Anthony Stentz, Herman Herman, Alonzo Kelly, Eric Meyhofer, G Clark Haynes, David Stager, Brian Zajac, J Andrew Bagnell, Jordan Brindza, Christopher Dellin, et al. Chimp, the cmu highly intelligent mobile platform. *Journal of Field Robotics*, 32(2):209–228, 2015.

[37] Ilkka Leppänen et al. *Automatic locomotion mode control of wheel-legged robots*. Helsinki University of Technology, 2007.

[38] Robin R Murphy. Rescue robotics for homeland security. *Communications of the ACM*, 47(3):66–68, 2004.

[39] K. Davies and A. Ramirez-Serrano. *A Reconfigurable USAR Robot Designed for Traversing Complex 3D Terrain*. Halifax, Nova Scotia, Canada, 2009.

[40] Krispin Davies. A novel control architecture for mapping and motion planning of reconfigurable robots in highly confined 3d environments. 01 2011.

[41] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 528–535, 2016.

[42] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[44] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[45] Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*, 2015.

[46] Zhou Zhou, Yan Xin, Hao Chen, Charlie Jianzhong Zhang, and Lingjia Liu. Pareto deterministic policy gradients and its application in 5g massive mimo networks. *ArXiv*, abs/2012.01279, 2020.

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

[48] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[49] Jack Collins, Shelvin Chand, Anthony Vanderkop, and Gerard Howard. A review of physics simulators for robotic applications. *IEEE Access*, PP:1–1, 03 2021.

[50] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Trans. Graph.*, 35(4), July 2016.

[51] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[52] John-Alexander M Assael, Niklas Wahlström, Thomas B Schön, and Marc Peter Deisenroth. Data-efficient learning of feedback policies from image pixels using deep dynamical models. *arXiv preprint arXiv:1510.02173*, 2015.

[53] Rituraj Kaushik, Pierre Desreumaux, and Jean-Baptiste Mouret. Adaptive prior selection for repertoire-based online adaptation in robotics. *Frontiers in Robotics and AI*, 6, Jan 2020.

[54] Blender Online Community. Blender—a 3d modelling and rendering package. *Blender Foundation*, 2018.

[55] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic. *ACM Transactions on Graphics*, 37(4):1–14, Aug 2018.

[56] Zhipeng Liang, Hao Chen, Junhao Zhu, Kangkang Jiang, and Yanran Li. Adversarial deep reinforcement learning in portfolio management, 2018.

[57] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[58] Savinay Nagendra, Nikhil Podila, Rashmi Ugarakhod, and Koshy George. Comparison of reinforcement learning algorithms applied to the cart-pole problem.

2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Sep 2017.

[59] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 9–20, 1998.

[60] Eric Krotkov, Douglas Hackett, Larry Jackel, Michael Perschbacher, James Pippine, Jesse Strauss, Gill Pratt, and Christopher Orlowski. The darpa robotics challenge finals: Results and perspectives. *Journal of Field Robotics*, 34(2):229–240, 2017.