University of Southampton

Faculty of Engineering and Physical Sciences

Electronics and Computer Science

Fast and Fluidic Adaptation of Robotic Locomotion to Damages and Environmental Changes

by

Yanislav Donchev

Fall 2020

Supervisor: Dr. Danesh Tarapore
Second Examiner: Prof. George Chen

A dissertation submitted in partial fulfilment of the degree

of MSc Artificial Intelligence

Abstract

The creation of robotic controllers that operate as reliably in uncontrolled outdoor environments as they do in laboratory conditions is a challenging task. Handcrafting such controllers is a robot-specific and a cost-ineffective procedure which does not generalise to unforeseen circumstances such as unanticipated terrains and damages. Recent work in the field focuses on learning-based techniques to automatically find such controllers. Although the potential of this approach seems promising, finding a solution that scales to complex problems and operates reliably without the need for human intervention is still an open question.

This dissertation investigates the ability to encode adaptable behaviours in a single neural network controller and realise non-episodic adaptation to damages and environmental changes. To achieve this, reinforcement learning agents are trained in diverse environments with gradually increasing complexity using automatic domain randomisation. Given proprioceptive sensor readings, three neural network agents based on feedforward, recurrent and convolutional architectures generate target joint angles for a hexapod robot. A terrain heightmap is provided as an extra input to the convolutional agent.

Our findings show that the memory-augmentation of the recurrent agent is crucial to the ability to adapt and the median performance of that agent drops by 50% on unstructured terrains with bumps, by 70% on structured terrains with stairs and by 30% on 18 unseen damages. The feedforward and convolutional agents fail to adapt reasonably to any of these situations. Automatic domain randomisation had a destructive effect to all agents and it did not encourage adaptation, possibly due to a low inductive bias in the architecture of the agents.

Overall, we provide a quantitative and a qualitative investigation on what matters in non-episodic adaptation and a partial solution to the problem. Further, we provide a modular codebase that enables easy implementation of future ideas in the field.

Acknowledgements

The use of first person plural in this dissertation signifies the collaborative thinking in this project. Discussions with Dr. Danesh Tarapore have inspired many of the design choices and methods and discussions with Dr. David Bossens shaped the results section.

My family and friends have been a constant source of joy and motivation, which made the writing of this dissertation go like a breeze.

Lastly, we acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

Statement of Originality

I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.

I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.

I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

I have acknowledged all sources, and identified any content taken from elsewhere.

I have used the following open-source software: *PyBullet*, *OpenAI Gym*, *rlpyt*, *PyTorch*, *OpenCV*, *OpenSimplex*, *NumPy*, *pandas*, *scikit-learn*, *SciPy*, and *Matplotlib*. Whenever relevant, the use of this software has been explained in the report and in the source files.

I did all the work myself, or with my allocated group, and have not helped anyone else.

The material in the report is genuine, and I have included all my data/code/designs.

I have not submitted any part of this work for another assessment.

My work did not involve human participants, their cells or data, or animals.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When robots leave the idealised lab conditions and meet unpredictable real-world scenarios, their reliability significantly decreases. Even when a robotic controller is designed with such situations in mind, it is impossible to anticipate everything that the robot might encounter. By contrast, animals in nature are agile and adaptive, being able to efficiently handle motion-related tasks in a variety of environments and with physical changes in their bodies (e.g. damages). Transferring these qualities to robots is an unsolved problem and the most robust attempts usually require meticulously designed hand-crafted gaits and painstaking fine-tuning. However, such methods are robot-specific, do not generalise to unseen environments and are cost-ineffective. Recent research in the field of robotics has been concentrated in automatically finding adaptive gaits using learning-based techniques such as evolutionary computation and reinforcement learning.

Classic hand-designed policies [1] are easy for interpretation and analysis and require little to no training. However, such policies are usually low-dimensional, which means that they are not expressive enough to scale to complex problems [2]. Furthermore, they are problem-specific and fail to adapt to damages in the robot and changes in the environment. By sacrificing interpretability, current state-of-the-art solutions learn a high-dimensional policy, usually a neural network, that maximises some objective function. Due to the poor performance of deliberative control methods in complex and dynamic environments, current research expands on the work of Brooks [3] and most control architectures implement a behaviour-based reactive control. One set of adaptation approaches is to fill a map with behaviourally diverse gaits and after deployment select a gait from that map that suits the environment [4–9]. These methods have all been successful in damage recovery but not without serious limitations. For example, the behavioural map is usually limited to a low dimensionality since its size and time to fill grow exponentially with the number of dimensions. Secondly, they find compensatory behaviours through trial and error which requires a human operator to manually reset the robot to the initial state after each try. Lastly, they are based on evolutionary computation which makes the learning of policies with millions of parameters computationally expensive. To overcome that last issue, another set of adaptation approaches make use of gradient-based reinforcement learning [10–14]. Although some of these approaches present the potential of emergent adaptation, their methods evaluate articulated bodies which are not physically realisable. And even when they are [14], the resulting gaits are slow.

Inspired by previous work in evolutionary computation and reinforcement learn-

ing, we aim to create a learning routine which allows robots to fluidically adapt to unseen environments and damages. Our approach uses proximal policy optimisation [15] as the reinforcement learning algorithm and automatic domain randomisation [16] as an aid to stimulate behavioural diversity. We evaluate three neural network policy architectures – feedforward, recurrent, and convolutional – to assess the contribution of each in adaptation. The structure for the rest of this dissertation is as follows: chapter 2 is an introduction to and a review of relevant literature related to evolutionary computation, reinforcement learning, meta-learning and domain randomisation; chapter 3 describes the learning routine in details; chapter 4 describes the experimental setup; chapter 5 and chapter 6 present our evaluation results on a variety of structured and unstructured terrains and unseen damages; lastly, chapter 7 is a conclusion and chapter 8 is a reflection on project management.

Chapter 2

Background and Theory

This chapter serves as an introduction to evolutionary computation, reinforcement learning and deep artificial neural networks and their application in robotic locomotion and adaptation. The end of this chapter discusses important topics on meta-learning and domain randomisation – both of which have found use in problems such as adaptation and generalisation.

2.1 Evolutionary Computation

Evolutionary computation (EC) [17] is a biologically-motivated approach to function optimisation. A candidate solution is referred to as an *individual*, which is described by a *genotype*. The ability of the individual to solve a problem is measured by its *fitness*, which the EC algorithm tries to maximise. EC is a population-based technique in which a set of candidate solutions is iteratively perturbed by small random changes (see Figure 2.1). After each iteration, the individuals with the highest fitness are used to generate the new *generation* of candidates. The main methods of introducing random perturbations are: *mutation*, in which the parameters of the solution are slightly varied; and *crossover*, in which the parts of the parameters from two solutions are combined. The gradient which the genotype updates follow is implicitly defined in the fitness function. This means that EC algorithms are gradient-free and there is no need for the function under optimisation to be differentiable.

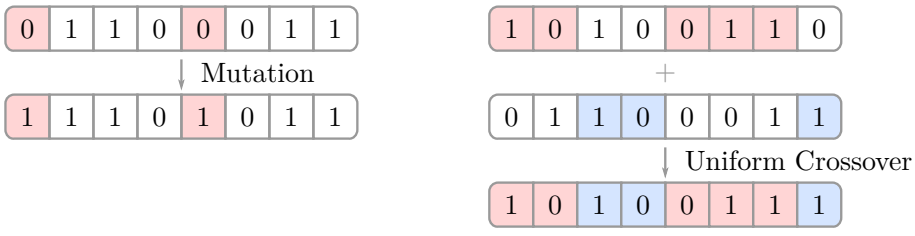


Figure 2.1: An example of the most common perturbations used in EC – *mutation* (left) and *uniform crossover* (right) – for an individual that is represented as an eight-bit string.

2.1.1 Quality Diversity Algorithms

A criticism of EC is that it only finds a single, potentially suboptimal solution, whilst evolution in nature also creates diversity. This has led to the study of quality diversity (QD) algorithms [18]. Instead of finding a single good solution, QD algorithms find a whole family of solutions, each of which is the best (referred to as an *elite*) in a particular niche. Quality diversity algorithms are actively studied to solve the problem of adaptation in robotics.

Multi-dimensional archive of phenotypic elites (MAP-Elites) [19] is arguably the most common quality diversity algorithm used in adaptation. The goal of the algorithm is to fill a discrete, low-dimensional map of elites, where each dimension represents a user-specified feature of interest. For example, a set of features could be the weight, height, and average power consumption of a robot. The MAP-Elites algorithm will find a high performing genotype for each combination of these features (e.g. light, tall and efficient; or heavy, short and less efficient). Some features, such as the weight and height, can be directly measured from the individual, whilst others, such as the average power consumption, need to be measured during the evaluation of the individual. After the user specifies a fitness function and a feature space of interest, MAP-Elites is initialised by generating a user-specified number of random genotypes. At every iteration, a genotype from the filled cells of the map is randomly chosen and is used to generate offspring via mutation and/or crossover. The features and performance of the offspring are then measured, and the cell in the map corresponding to the features is filled with the offspring if: 1) the cell is empty, or 2) the current genotype in the cell has lower performance than the offspring.

One of the early successes of MAP-Elites in fast damage recovery is its combination with Bayesian optimisation used as an intelligent trial-and-error (IT&E) algorithm [4]. The first step in the IT&E algorithm is the creation of a behaviour-performance map using MAP-Elites. This map reflects the robot's knowledge of different behaviours and their performance. If a drop in performance is detected after deployment, the second step of the algorithm is triggered: a search of the map for a compensatory behaviour using Gaussian process regression [20], in which the performance predictions from the behaviour-performance map are used as a prior. The IT&E algorithm could successfully find compensatory behaviours to a damaged[1] hexapod robot in a handful of trials. Although the robot had not encountered any of these damages during simulation, the features in MAP-Elites were the duty factor of each leg (i.e. the proportion of time that each leg is in contact with the ground). Instead of explicitly searching for different behaviours, the quality-environment-diversity (QED) algorithm [5] attempts to achieve behavioural diversity by creating environmental diversity of the environment in which the agents learn. The learnt environment-performance map allowed for damage recovery from injected faults, which have not been shown during evolution.

One of the problems with MAP-Elites is that the number of cells in the map grows exponentially with the number of dimensions, which makes it impractical for large problems since it would take exponentially more memory and time to fill the whole map. An extension to the algorithm, called centroidal Voronoi tessellation MAP-Elites [6] divides the features space using a centroidal Voronoi tessellation [21] rather than a grid, which allowed for a 1000-dimensional feature space without a

[1] The authors tested broken, missing and unpowered legs.

compromise in performance when tested on a hexapod locomotion task. Another disadvantage of all algorithms mentioned so far is that they need to perform trial and error to find a compensatory behaviour. By introducing a *generic reward* to the IT&E algorithm a [7], a semi-episodic recovery is possible in which the robot updates its behaviour whilst executing its task. Another approach [8], uses a Monte-Carlo tree search [22] to update the robot's behaviour every three seconds in an attempt to solve the problem of reset-free damage recovery. Nevertheless, these algorithms still need to perform multiple trials until a satisfactory behaviour is found, which can potentially damage a deployed robot.

Whereas recent studies in robotics seek to optimise deep neural networks with millions of parameters, most studies on robotic adaptation that use MAP-Elites [4, 7–9, 23] use simple, low-dimensional representations as their solutions. The reason is that – although not impossible [24] – training such networks using EC is computationally expensive. Colas *et al.* [25] propose a modification to MAP-Elites that uses *evolutionary strategies* [26] instead of evolutionary computation as the underlying optimisation algorithm. Their algorithm successfully trains a biped and a quadruped in a range of tasks but its performance is yet to be explored on real robots, environments and damages. Apart from this modification to MAP-Elites, the traditional way of training deep neural networks is reinforcement learning.

2.2 Reinforcement Learning

The goal of the learner in reinforcement learning (RL) is to approximate an optimal behaviour – that is, to learn how to act optimally in the current situation – in a way that maximises a numerical reward which reflects its success or failure. As it will become apparent later in this section, reinforcement learning is widely used to solve problems in robotics, although the same concepts apply to achieve human-level performance in Atari games from raw pixel values [27], drug design [28], and computing resources allocation [29]. Since reinforcement learning forms a substantial part of this dissertation, the rest of this section will serve as an introduction to the topic.

2.2.1 Markov Decision Processes

The problems solved with reinforcement learning can be represented as a sequential decision-making processes. The decision-maker is referred to as the *agent*, and everything outside the agent is the *environment*. Figure 2.2 shows an intuitive overview of the continual interaction between the agent and the environment at discrete timesteps $t \in \{1, 2, 3, ...\}$. The agent picks an action and the environment responds by emitting a new state[2] and a numerical reward which has the role of a feedback signal. The goal of the agent is to maximise the accumulated reward over time by appropriately changing its actions.

More formally, the problem can be presented as a Markov decision-making process (MDP) [30], which is a tuple $\langle \mathcal{S}, \mathcal{A}, r, p, p_0 \rangle$ where [31]: $\mathcal{S}$ is the set of all possible states; $\mathcal{A}$ is the set of all possible actions; $r : S \times A \times S \mapsto \mathbb{R}$ is the reward function;

[2]Note that the state is a superset of the observation that the agent actually "sees". However, both terms – state and observation – are often used interchangeably in the literature.

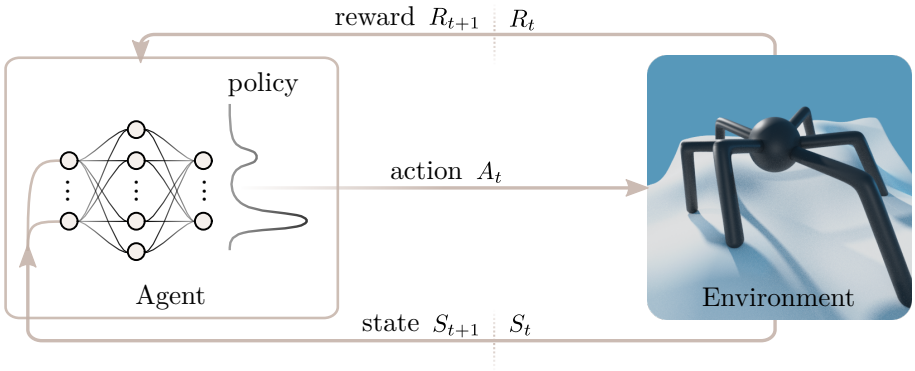


Figure 2.2: The continual agent-environment interaction through states, actions, and rewards.

$p : S \times A \mapsto Pr(S)$ is the transition probability function which models the dynamics of the environment; p_0 is a distribution over the initial states.

At every timestep t, the environment generates a state $S_t \in \mathcal{S}$. The agent observes the state and picks an action $A_t \in \mathcal{A}$. The environment then emits a new state $S_{t+1} \in \mathcal{S}$ and a reward signal $R_{t+1} = r(S_t, A_t, S_{t+1}) \in \mathbb{R}$. For episodic tasks[3], this process is repeated until a terminal state is reached at $t = T$. The generated sequence of states, actions and rewards is called a *trajectory* $\tau = S_0, A_0, R_1, S_1, A_1, ..., S_T$.

Given any state s and an action a, each new environment state s' and reward r are sampled from a probability distribution (Equation (2.1)) which defines the environment dynamics. Agents which have access to, or learn this distribution are called *model-based* ([32] is a notable example). A model-based approach could offer the ability to plan future actions and increase sample efficiency, however, a model of the environment is rarely available and *model-free* methods are the popular choice in literature [33]. The following subsections and the rest of this dissertation will be based on model-free reinforcement learning.

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

The ultimate goal of the reinforcement learning agent is to maximise the *expected return*. The return G_t is defined as the sum of the discounted future rewards $\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$, where $\gamma \in [0, 1]$ is the discount factor. The use of a discount factor enables control over the present value of future rewards.

Policies and Value Functions

The agent selects actions by following a policy, denoted as π. The policy is a mapping from states to actions. If the policy is deterministic, $\pi(s)$ denotes the action taken in state s. If the policy is stochastic, $\pi(a|s)$ denotes the probability of taking action a in state s. Crucial elements in nearly all reinforcement learning algorithms are *value functions*. Given a state or a state-action pair, value functions estimate the expected return the agent will receive by following a particular policy starting from that state (or state-action pair). The value functions as a function of a state and a state-action pair are defined in Equations (2.2) and (2.3) respectively.

[3]Tasks, which naturally complete at some timestep, as opposed to continue forever.

$$v_\pi(s) \doteq \mathbb{E}\left[G_t | S_t = s\right] \tag{2.2}$$
$$q_\pi(s,a) \doteq \mathbb{E}\left[G_t | S_t = s, A_t = a\right] \tag{2.3}$$

The goal of reinforcement learning is to find an (approximately) optimal policy π_*. Value functions provide a natural way of comparing policies by giving them a partial order; a policy π is at least as good as a policy π' (denoted by $\pi \succeq \pi'$) if and only if $v_\pi \geq v_{\pi'}$.

Exploitation and Exploration

Exploitation is the behaviour when the agent is acting greedily. In other words, in every state, it is taking the action with the highest value. *Exploration*, on the other hand, is the behaviour when the agent is selecting apparently inferior actions. One of the keys to learning a near-optimal policy is that of balancing exploitation and exploration – an agent that is mostly exploiting might get stuck on a local optimum, while an agent that is mostly exploring might never converge to a solution [31].

The ϵ-greedy method of balancing exploration and exploitation picks a random action with probability $\epsilon \in [0, 1]$ and acts greedily otherwise. A major drawback with ϵ-greedy is that there is always some probability of picking a random action. As it will become apparent in §2.2.3, more robust methods use stochastic policies which output the parameters of a probability distribution (e.g. a mean and a standard deviation). This distribution can then be sampled to choose an action, which realises exploitation. An advantage of this is that as the agent becomes more certain over the state values, these distributions shrink and the policy approaches determinism [31].

2.2.2 Tabular and Approximate Action-Value Methods

The choice of a reinforcement learning algorithm is strongly dependent on the type and scale of the state and action spaces and how the policy is represented. In the simplest case, the state and action spaces are discrete in which case the policy can be represented as a table which stores the value of each state-action pair. By learning this table, the agent will know the value of each action at every state and would use this knowledge to decide how to behave. Algorithms of this kind are called *action-value methods* since the policy is completely defined by the action-value estimates. The fact that the information is stored in a table, makes them *tabular methods*. Tabular algorithms, such as Q-learning [34], have shown to be successful in complicated planning tasks [35, 36].

Nevertheless, storing a table would not scale with big state spaces. Even with unlimited memory, it is unlikely that the agent will see every state and be able to fill the whole table. The *approximate methods* use function approximation techniques (introduced in §2.3) from *supervised learning* to generalise to a large number of states, by only encountering a small part of all possible states [27].

2.2.3 Policy Gradient Methods

All methods discussed yet are action-value methods which means that for each action, the agent estimates a value and chooses an action correspondingly (e.g. the

highest value action if acting greedily). This property of action-value methods makes them unsuitable to problems with continuous action spaces, such as robotic control, since there are infinitely many actions. *Policy gradient methods* use a parameterised stochastic policy π_θ, where $\boldsymbol{\theta}$ is the policy's parameter vector. This vector is iteratively updated by following the gradient – scaled by $\alpha \in \mathbb{R}$ – of the expected return $J(\boldsymbol{\theta})$ under the current policy:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_\theta J(\boldsymbol{\theta}_t)\big|_{\boldsymbol{\theta}_t} \tag{2.4}$$

A key equation to policy gradient methods is the advantage function in Equation (2.5). The advantage of an action is a relative measure which reflects how much better it is to take a certain action under the policy π, compared to randomly selecting an action. Policy gradient algorithms aim to increase the probability of selecting actions with a high advantage. Although the advantage function is not known in practice, it can be estimated using generalised advantage estimation (GAE) [37].

$$A_\pi(s, a) \doteq q_\pi(s, a) - v_\pi(s) \tag{2.5}$$

To the best of our knowledge, all recent work on reinforcement learning and robotics is based on policy gradient methods and uses a neural network in some way as the policy architecture. Most of the early research on continuous control in robotics and locomotion [10, 11, 15, 37–39] has been mainly concerned with the emergence of a locomotor behaviour. Nevertheless, some of this work has shown that by extracting features from the surrounding terrain through a neural network [10] and building hierarchical networks for control and planning [39], the reinforcement learning agent can learn to navigate through complex terrains.

Using a neural network as the agent's policy offers the flexibility of learning complex representations and non-obvious relationships between the state and the action spaces. However, this flexibility results in a low *inductive bias*, which makes the learning slow, and the resulting gaits look unnatural. Peng *et al.* [40] introduce an example-guided learning process, in which the agent is tasked to mimic a reference motion, whilst also accomplishing its primary task. This is achieved by adding imitation components to the reward function; these are a function of the mean squared error between the joint angle positions, joint angle velocities, the centre of mass, and the end-effector positions of the robot and a pre-recorded reference gait. An extension to this work [12] transfers real animal gaits to a robot with a different morphology. Although these mimic-based methods produce naturally looking gaits, the generalisation of their applicability is questionable since animals usually greatly exceed the degrees of freedom in a robot. Abdolhosseini *et al.* [41] discuss the concept of using *mirrored trajectories* to generate a policy which would behave in the same way if all sensors and actuators of an articulated body were swapped along its axis of symmetry. The result is a symmetric gait. Although symmetry is expected and desirable most of the time, its use is uncertain when a damage breaks the symmetry in the robot, or when a terrain – such as a left-to-right slope – is encountered.

Apart from terrain navigation and locomotor behaviour quality, reinforcement learning techniques have been studied for terrain adaptation. Peng *et al.* [11] train multiple actors and critics in a single network. After each locomotion cycle, the quality of each actor for the terrain is predicted, and that actor is used for the rest

of the cycle. In this way, different actors can specialise to different types of terrains. However, this adaptation step cannot occur in the middle of a locomotion cycle. Azayev *et al.* [13] train multiple expert policies on a variety of terrains; a long short-term memory neural network is trained offline to categorise each type of terrain from the observations of the agent. At deployment, the long short-term memory neural network tries to predict the terrain type and selects the corresponding expert policy. A shortcoming of both these methods is that they assume that the world can be modelled by a discrete number of terrain types; out-of-distribution evaluation is required to solidify their findings. Tsounis *et al.* [14] also explore a hierarchical approach. A gait planner network generates desired feet locations based on a visual heightmap and proprioceptive sensors, whilst a gait control network attempts to achieve these locations and maintain balance at the same time.

Most work in continuous control is based on adaptations of soft actor-critic [42], proximal policy optimisation (PPO) [15], and asynchronous advantage actor-critic [43]. The main trade-offs when considering between these algorithms are sample-efficiency, simplicity, time-complexity and stability. Despite not being the most sample-efficient, PPO is the most widely adapted algorithm due to its simplicity, ease of parallelisation and stability to hyperparameter choices.

2.2.4 Proximal Policy Optimisation

Proximal policy optimisation [15] – the successor of trust region policy optimisation [38] – is an on-policy, policy gradient, actor-critic method, which means that it optimises a stochastic policy together with a value function estimator. The main problem that PPO tries to solve is that of taking big improvement steps, while not deviating much from the original policy which could be detrimental to performance. It achieves this by a special clipping function which prevents big deviations between policy updates. The simplicity, stability and good performance of PPO have made it a popular choice in literature [10, 12–14, 40, 41].

In PPO, the old policy π_{old} is used to collect a set of trajectories by running it in the environment. To speed up the learning process, the trajectories can be collected from many environments in parallel. Once the trajectories are collected, the advantages at each time-step are estimated using generalised advantage estimation. The policy is then updated through multiple epochs of stochastic gradient ascent to maximise the PPO objective[4] in Equation (2.6) where ϵ is the clipping ratio parameter and $\hat{A}_{\pi_{old}}$ is the estimated advantage. The intuition behind this objective is that, in general, a state-action pair with a negative advantage will reduce $\pi(a|s)$ unless it is smaller than $(1-\epsilon)\ \pi_{old}(a|s)$; a positive advantage will increase $\pi(a|s)$ unless it is greater than $(1+\epsilon)\ \pi_{old}(a|s)$. This way of clipping the objective allows for multiple epoch of gradient ascent without destroying the policy.

$$L_{PPO}(s,a,\pi,\pi_{old}) = \mathbb{E}\left[\min\left(\frac{\pi(a|s)}{\pi_{old}(a|s)}\hat{A}_{\pi_{old}}, \text{clip}\left(\frac{\pi(a|s)}{\pi_{old}(a|s)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_{\pi_{old}}\right)\right] \quad (2.6)$$

[4]The original PPO paper introduces two objectives – PPO-Clip and PPO-Penalty – however, the former is shown to outperform the latter and so is used in this dissertation (also, it is the preferred in literature).

2.3 Deep Artificial Neural Networks

As mentioned in §2.2.2 and §2.2.3, reinforcement learning problems with large state spaces and continuous action spaces require some form of differentiable function approximation. Deep neural networks (DNNs) are universal function approximators which are usually trained by back-propagating errors with gradient descent [44]. In recent years, the use of DNNs in the offline supervised setting has set the state-of-the-art in many areas such as image classification [45] and natural language processing [46]. Their universality has found its way into robotic control and online learning. This section will introduce the three most common network types and their applications.

2.3.1 Feedforward Networks

Feedforward networks form the basis of deep learning. The goal of a feedforward network is to approximate a function $f^*(\cdot)$ by learning the values of a parameter vector $\boldsymbol{\theta}$, such that $f(\cdot, \boldsymbol{\theta}) \approx f^*(\cdot)$ [47]. They are called feedforward since the flow of information is only in one direction without loops. The term "deep" in deep neural networks comes from the fact that neural networks are a stack of layers – the number of layers is the depth. A single feedforward layer multiplies its input by a weight matrix W, adds a bias vector $\boldsymbol{b}$ and processes the result through some non-linear function $g(\cdot)$ – the expression of such layer is $f^{(i)}(\boldsymbol{x}, W, \boldsymbol{b}) = g(\boldsymbol{x}W^T + \boldsymbol{b})$. The chaining of these layers results in the full network, for example $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. Apart from being the building block for more complex models, feedforward networks are ubiquitous in robotic control [5, 10–16, 25, 37, 39–41, 48–50].

2.3.2 Convolutional Networks

Convolutional neural networks (CNNs) [51] are a special type of feedforward networks in which at least one of the layer operations is *convolution* as opposed to matrix multiplication. The output of a convolutional layer is the convolution between the *input* and a *kernel*, optionally passed through a non-linear function. Two features of CNNs make them particularly efficient for problems in which the data has a grid-like structure – for example images or sound. Firstly, since the convolution operation is applied to a patch of nearby values of the input, CNNs are able to combine neighbouring sources of information. Secondly, since the same kernel is applied to all patches within an input, the number of learnable parameters is greatly reduced; furthermore, the learning machine becomes invariant to position. Convolutional neural networks are commonly used in robotic control to extract visual features from a camera or a local heightmap [11, 14, 16, 48, 50].

2.3.3 Recurrent Networks

One shortcoming of both network architectures from above is that they do not store representation of recently seen inputs which is problematic when working with sequential data. Recurrent neural networks (RNNs) aim to solve this problem by adding a feedback loop to the standard feedforward networks. The recurrence can be formally defined by $\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}, \boldsymbol{\theta})$, where $\boldsymbol{h}$ represents the *state* of the layer

[47]. This kind of feedback allows for short-term memory, which can be significant in non-Markovian control [52]. A problem with standard RNNs is that "exploding" or "vanishing" gradients make them unsuitable for long-term dependencies; the introduction of gating units in long short-term memory (LSTM) networks allows for controlled gradient flow and the learning of long-term dependencies [53]. In robotic control, LSTM networks have been used to solve non-Markovian problems and aid meta-learning [13, 16, 50].

2.4 Meta-Learning and Domain Randomisation

Meta-learning – or *learning to learn* – has been studied in the fields of EC and RL as a manner to significantly reduce the number of trials needed to train an agent [54, 55], to bridge the simulation-reality gap [16] and automatically discover behavioural features that maximise adaptation [56]. The type of meta-learning that we are interested in is that of implicitly learning to learn the dynamics of the environment, while the actual optimisation task is to learn to walk. Learning the dynamics of the environment would allow the agent to predict future states based on the current state and adapt appropriately. Previous work has shown that domain randomisation is key to adapting to unseen situations and inferring knowledge about the environment without explicitly learning it [5, 16, 48–50]. Akkaya *et al.* [16] show that adding memory to the learning agent creates an internal learning algorithm that can adjust the agent's behaviour and improve its performance in the current task.

In the following experiments, we use automatic domain randomisation (ADR) [16] to create a learning curriculum for the agent that gradually complexifies the training environment as the agent learns to solve it. The rest of this section will introduce ADR conceptually – a formal definition and implementation details are available in §3.2.1. The ADR algorithm assumes that the environments can be described by a finite number of parameters which are sampled from a uniform distribution – the narrower that distribution is, the easier the environments are. ADR operates independently from the training algorithm – the combination of ADR and the training algorithm leads to the training flow in Figure 2.3. In a loop, the training procedure goes from sampling environments from the ADR distribution, training the agent on these environments and then changing the distribution sizes according to the agent's performance.

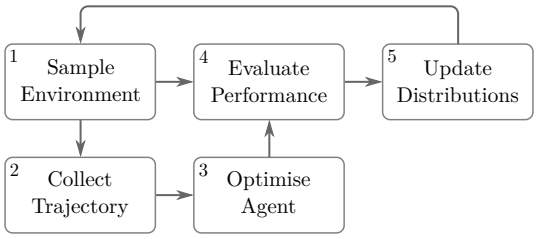


Figure 2.3: Overview of ADR. After sampling from a distribution over environments (1) and collecting a trajectory (2), a policy is optimised (3) and evaluated on the current distribution over environments (4). The value of the evaluation performance is used to decide whether to shrink, expand or leave unchanged this distribution (5). (Adapted from [16])

Chapter 3

Methodology

3.1 Policy Architectures

As mentioned in §2.2.3, the policies in this work are represented by neural networks. Three types of networks are assessed and implemented using *PyTorch*. A standard fully-connected (FC) network is used as a general function approximator and a baseline for comparison. A long short-term memory (LSTM) network is used as a mechanism that allows meta-learning and helps with solving partially observable MDPs. Lastly, a convolutional network (CNN) is used as a method to extract visual features from a heightmap of the terrain and use them for control. The three architectures are described in detail below. Many of the design decisions below have been based on a large-scale empirical study [57] which analysed the effects of hyperparameter choices on the performance of policy-gradient reinforcement learning on robotic locomotion tasks. Any other choices which have been based on other work or our decisions are explicitly highlighted below.

3.1.1 Fully-Connected Policy

The actor-critic FC policy is shown in Figure 3.1. The inputs to the network are the proprioceptive observations O_P described in §4.3. The actor and the critic do not share parameters, but their architectures are the same. A *tanh* non-linearity is used between the hidden layers, and the mean output vector of the actor is additionally processed by the *tanh* function. The standard deviation vector in the actor is independent of the observations and is initialised at 0.7 – this reduction from the default value of 1.0 makes action selection less random and makes learning faster

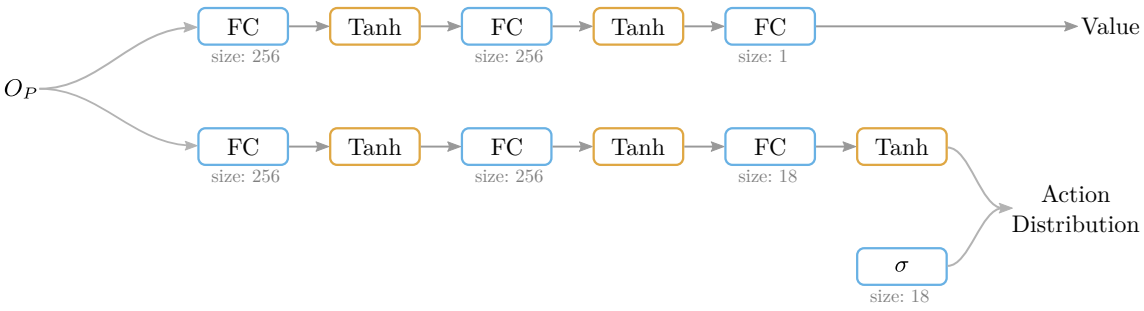


Figure 3.1: The FC policy network.

and more stable [57]. The FC layers are initialised using the method described in [58]. The initialisation values in the last layer of the actor are scaled by 0.05 which makes the actions in the beginning of training less dependent on the state.

3.1.2 Recurrent Policy

The recurrent policy network is exactly the same as that in the FC policy, but the middle hidden layer is LSTM rather than FC. This type of network has been successfully used in [16] and it has a similar number of learnable parameter as the network in the FC policy, which makes the two architectures comparable.

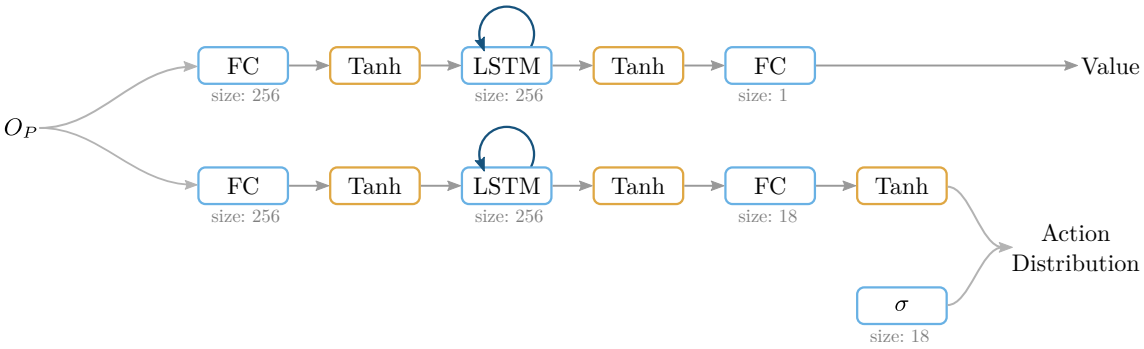


Figure 3.2: The LSTM policy network.

3.1.3 Convolutional Policy

The convolutional policy network is essentially the same as the FC policy network, but it has 64 additional features appended to the proprioceptive observations O_P. These features are generated by the CNN branch of the policy network, which processes the heightmap observation O_H (described in §4.3). This branch consists of three convolutional layers with rectified linear unit (ReLU) non-linearities, and a final fully connected layer with another ReLU non-linearity. The task of this network is to extract features from the heightmap (e.g. distance to a step or position of an obstacle), which could be useful for coordination and control of the robot. A

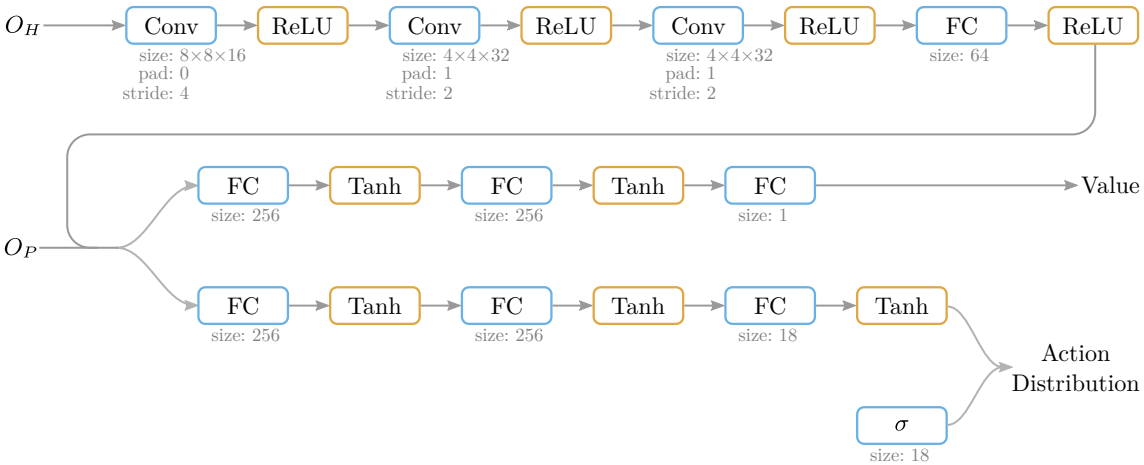


Figure 3.3: The CNN policy network.

similar network architecture has been successfully used to navigate a robot around obstacles [39, 40].

3.2 Implementation of Training Algorithm

This section summarises the implementation of ADR, how it was combined with PPO, and how the training algorithm is parallelised to run multiple trajectory sampling workers simultaneously.

3.2.1 Automatic Domain Randomisation

The following formalisation is based on [16]. In ADR, a set of parameters $\lambda \in \mathbb{R}^d$ is used to generate a parameterised environment e_λ with d randomised parameters. Each parameter $\lambda_i,\ i \in \{1, ..., d\}$ is sampled from a uniform distribution $\lambda_i \sim \mathcal{U}(\phi_i^L, \phi_i^H)$, where ϕ_i^L and ϕ_i^H are the low and high boundaries respectively. Together, all boundaries $\phi \in \mathbb{R}^{2d}$ give rise to the distribution in Equation (3.1), in which ϕ is allowed to change according to the agent's performance.

$$P_\phi(\lambda) = \prod_{i=1}^{d} \mathcal{U}\left(\phi_i^L, \phi_i^H\right) \tag{3.1}$$

The ADR entropy in Equation (3.2) provides a measure to quantify the expansion of the distribution over environments; it is measured in nats[1]/dimension. A high ADR entropy reflects a high randomisation of the generated environments.

$$\mathcal{H}(P_\phi) = \frac{1}{d} \sum_{i=1}^{d} \ln\left(\phi_i^H - \phi_i^L\right) \tag{3.2}$$

At initialisation, the initial distribution over environments P_{ϕ_0} is defined by boundaries ϕ_0; this distribution is narrow and might even be concentrated at a single point. During training, the ADR algorithm is realised by the *sample* and *update* procedures below.

The Sample Procedure

The sample procedure in Algorithm 1 is responsible for sampling the ADR distribution over environments P_ϕ and generating an environment based on the sampled parameter vector λ. In addition, with a boundary sampling probability p_b, the procedure samples the low or high boundary of one of the environment parameters. If a parameter boundary has been sampled, the procedure returns information about its index and whether the *low* or *high* boundary has been sampled. Restricting boundary sampling with the boundary sampling probability allows the agent to learn on easier environments for longer – as opposed to always having one of the parameters on its hardest value.

Algorithm 1 ADR environment sampling – a procedure for sampling an environment from an ADR distribution P_ϕ.

procedure AdrSample(P_ϕ)
  $\lambda \sim P_\phi$ ▷ Sample environment parameters from the distribution
  $x \sim U(0, 1)$
  if $x \leq p_b$ **then** ▷ Sample boundary for evaluation with probability p_b
    $i \sim U\{1, ..., d\}$, $x \sim U(0, 1)$ ▷ Sample a parameter index
    if $x \leq 0.5$ **then** ▷ Sample the low boundary
      $b \leftarrow$ low ▷ Store the sampled boundary in variable b
      $\lambda_i \leftarrow \phi_i^L$ ▷ Overwrite parameter λ_i with low boundary.
    else ▷ Sample the high boundary
      $b \leftarrow$ high ▷ Store the sampled boundary in variable b
      $\lambda_i \leftarrow \phi_i^H$ ▷ Overwrite parameter λ_i with high boundary.
    end if
  else
    $i, b \leftarrow \emptyset$ ▷ No parameter boundary is being evaluated
  end if
  return e_λ, i, b ▷ Return env. and sampled boundary information (if any)
end procedure

Algorithm 2 ADR distribution updating – a procedure for updating the boundary $\langle i, b \rangle$ of an ADR distribution P_ϕ, given evaluation performance p.

procedure AdrUpdate(P_ϕ, i, b, p)
  if $i, b \neq \emptyset$ **then** ▷ If a boundary has been evaluated
    if $b =$ low **then** ▷ If it was the low boundary
      $D \leftarrow D_i^L$, $\phi \leftarrow \phi_i^L$ ▷ Get performance buffer and boundary value
    else if $b =$ high **then** ▷ If it was the high boundary
      $D \leftarrow D_i^H$, $\phi \leftarrow \phi_i^H$ ▷ Get performance buffer and boundary value
    end if
    $D \leftarrow D \cup p$ ▷ Append performance to the buffer
    if $Length(D) \geq m_i$ **then** ▷ If the buffer is full
      $\bar{p} \leftarrow Mean(D)$, $D \leftarrow \emptyset$
      if $\bar{p} \geq t_i^H$ **then** ▷ If the mean performance is acceptable
        $\phi \leftarrow \phi + \Delta_i$ ▷ Expand parameter distribution
      else if $\bar{p} \leq t_i^L$ **then** ▷ If the mean performance is unacceptable
        $\phi \leftarrow \phi - \Delta_i$ ▷ Shrink parameter distribution
      end if
    end if
  end if
end procedure

The Update Procedure

The update procedure in Algorithm 2 is responsible for updating the value of a boundary that has been selected for evaluation in Algorithm 1, based on the achieved

[1]The nat is a unit of entropy obtained by using the natural logarithm. It is equivalent to the bit which is obtained from a base-2 logarithm.

Algorithm 3 Trajectory worker – collects a trajectory of size M by running the latest policy in the environment. The lines in blue highlight how ADR is integrated into a standard trajectory worker.

Get policy parameters θ from shared memory
Get ADR distribution P_ϕ from shared memory
$\langle e_\lambda, i, b\rangle \leftarrow$ AdrSample(P_ϕ) ▷ Sample initial env. from ADR distribution
Observe the initial state s in environment e_λ
repeat
    Initialise empty trajectory $\tau \leftarrow \emptyset$
    for timestep $\in [1, ..., M]$ **do** ▷ Sample M timesteps from the environment
        $a \sim \pi_\theta(a|s)$ ▷ Sample action from the latest policy network
        Take action a, observe state s' and reward r
        Push (s, a, r) to trajectory τ
        $s \leftarrow s'$
        if s is terminal **then** ▷ Update ADR distribution and reset environment
            $p \leftarrow 1$ if end of terrain is reached, else 0
            AdrUpdate(P_ϕ, i, b, p)
            $\langle e_\lambda, i, b\rangle \leftarrow$ AdrSample(P_ϕ)
            Observe the initial state s in the new environment e_λ
        end if
    end for
    yield the collected trajectory τ
until training is finished

performance p of the agent. Each boundary has its own performance buffer D of maximum size m which gets filled with the agent's performance when that boundary has been sampled for evaluation. Once the buffer fills up, the mean performance $\bar{p}$ is calculated – if $\bar{p}$ is greater than some upper threshold t^H the boundary is expanded by Δ; if $\bar{p}$ is less than some lower threshold t^L the boundary is shrunk by Δ; for any other value of $\bar{p}$ the boundary remains the same.

3.2.2 Proximal Policy Optimisation

Integrating ADR into proximal policy optimisation does not require a modification of the typical PPO implementation in [15]. However, a minor modification of the trajectory workers is required. A trajectory worker is a subprocess which runs the agent in the environment and collects trajectory data which is then used in PPO. The blue lines in Algorithm 3 show the required additions that integrate ADR. Whereas a normal trajectory worker would simply reset an environment that has terminated, the trajectory worker in Algorithm 3 also measures the performance of the agent, updates the ADR distribution and samples a new environment from it. The only addition to PPO in Algorithm 4 is the initialisation of the ADR distribution parameters in shared memory.

Parallel Implementation

Figure 3.4 illustrates the parallel implementation of the training algorithm using Python multiprocessing and *rlpyt* [60]. The master process coordinates PPO for

Algorithm 4 Proximal policy optimisation – Implementation of PPO with the Adam optimiser [59]. The lines in blue highlight the integration of ADR.

1: $\theta \leftarrow$ random weights ▷ Initialise actor network weights in shared memory
2: $\psi \leftarrow$ random weights ▷ Initialise critic network weights in shared memory
3: $P_\phi \leftarrow P_{\phi_0}$ ▷ Initialise ADR distribution in shared memory
4: **for** $k \in [1, ..., K]$ **do** ▷ Repeat K times.
5: $D \leftarrow \{\tau_1, ..., \tau_N\}$ ▷ Collect trajectories from N trajectory workers
6: Compute advantages $\hat{\boldsymbol{A}}$ for trajectories in D using latest critic V_ψ and GAE
7: Optimise L_{PPO} wrt θ using Adam for L epochs and minibatch size M
8: Optimise V_ψ by regression on the mean-squared error of value estimates and rewards-to-go using Adam for L epochs and minibatch size M
9: **end for**

updating the agent and a sampler for sampling trajectories from each trajectory worker. The agent parameters and the ADR distributions are initialised in shared memory and are synchronised across workers after each batch of trajectories. Each trajectory worker holds a copy of the agent, which runs on the CPU during trajectory sampling. Once all trajectories have been sampled, they are copied onto the GPU where the agent is optimised. This configuration minimises the communication between processes and achieves better wall-clock time than the alternative in which the agents used for sampling are on the GPU.

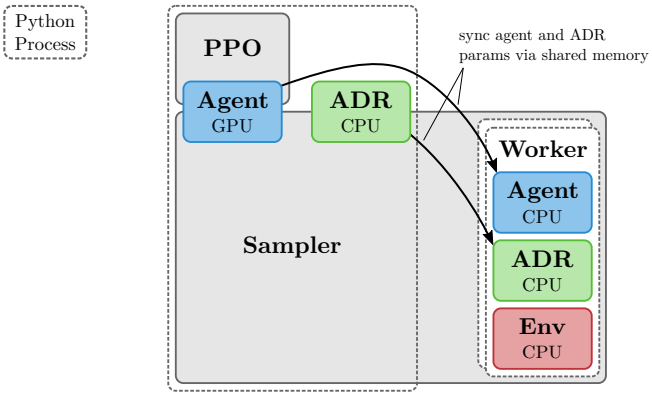


Figure 3.4: Parallel implementation of the training algorithm. Agents and environments execute in parallel processes. Optimisation is done on the GPU. (Adapted from [60])

Chapter 4

Experimental Design

4.1 Hexapod Robot Model

The typical robot morphologies used in learning-based methods for finding locomotion gaits include bipedal [10, 39, 61], quadrupedal [14, 62] and hexapedal [4, 7, 8, 13, 23] robots. In this work, we use a hexapedal robot since its wide morphology makes it inherently stable, and the high degree of redundancy is likely to allow the robot to continue operating even after a damage has occurred. The robot is the Pexod [63] which has a total of 18 degrees of freedom (DoF) – three per leg (see Figures 4.1a and 4.1b). The revolute joints of the robot are realised by the Dynamixel MX-28 servos [64]. Servos s_1^i[1] attach the legs to the body and are responsible for moving the legs horizontally in the range $[-\pi/8, \pi/8]$ rad. Servos s_2^i and s_3^i move the legs vertically in the range $[-\pi/4, \pi/4]$ rad. The robot is simulated using PyBullet [65] with a simulation frequency of 240 Hz and a control frequency of 60 Hz. The actuator parameters are tuned according to the specification [64] of the servos. Each actuator is driven to the desired joint angle by a proportional–integral–derivative (PID) controller, which is internal to the servos.

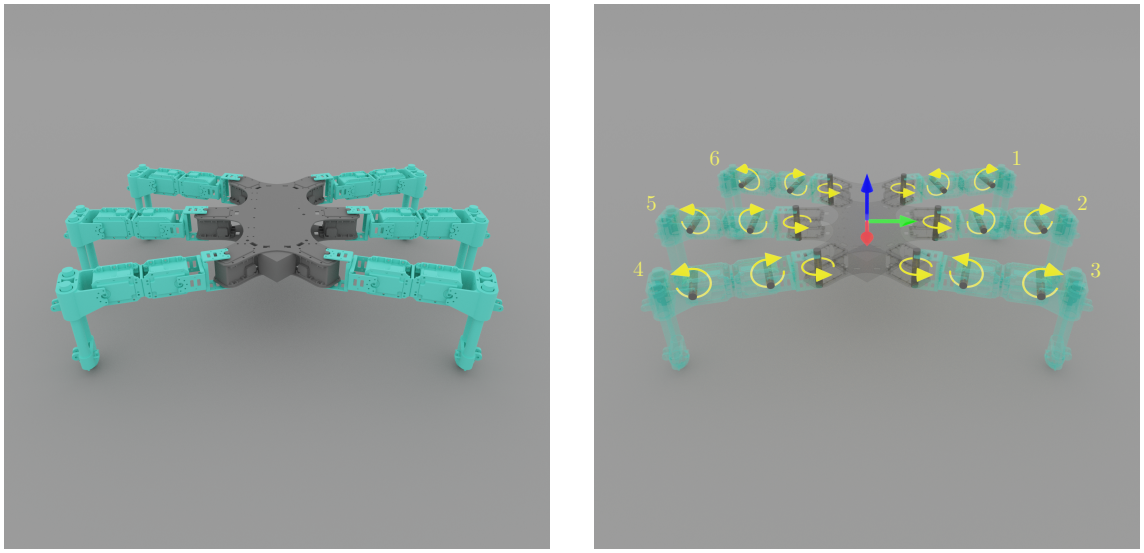
4.2 Reference Controller

A reference controller that was found by the MAP-Elites algorithm [19] is used as a benchmark. Although that controller was found as a part of the map in the IT&E algorithm [4], it is not adaptive since the adaptation step of IT&E is not used in our experiments. To generate target joint angles, the controller outputs a 1 Hz square wave, convolved with a Gaussian kernel which smooths the wave. Each square wave i is parameterised by its amplitude α_i, phase ϕ_i and duty cycle τ_i. This controller reduces the effective DoF of the robot from 18 to 12 by always setting the target angle of servos s_3^i to be the same as those of s_2^i.

4.3 Observation Space

The observation space defines how the agent sees the environment; in other words, what are the inputs to the control policy. The higher the dimensionality of the

[1]Here $i \in [1, ..., 6]$ refers to the index of the leg which is attached to the servo.



(a) The Pexod hexapod robot. (b) Joint description and leg numbers.

Figure 4.1: Rendering of the Pexod hexapod robot and its joints. The robot definition was taken from the Resibots repository [63].

observation space, the less information the agent has to infer. However, a big observation space could contain information that is irrelevant to the problem and slow down convergence. We chose the observation space based on previous work [11, 13, 14, 39], but also considered whether the added sensors could realistically be available on a physical robot. All components of the observation space are in the body frame of the robot, which makes the learnt policy invariant to position.

4.3.1 Sensors

The observation space $O_P = \langle O_{\hat{q}}, O_v, O_\omega, O_a, O_\alpha, O_y, O_g, O_\theta, O_{\dot{\theta}}, O_\tau, O_F \rangle$ with all proprioceptive sensors included is a 113-dimensional vector (the individual components are defined below). The policies with a visual input receive an additional 32×32 heightmap matrix O_H which makes the observation space a 1137-dimensional vector. The following section introduces the components of the observation space, their dimensionality and, wherever relevant, how they can be obtained in a physical system.

Body orientation $O_{\hat{q}} \in \mathbb{R}^4$: A quaternion, which describes the orientation of the robot's body with respect to the world frame. A quaternion representation was used, rather than Euler angles, since it does not suffer from singularities [1]. Orientation can be estimated by processing accelerometer, gyroscope and magnetic field sensor data with an orientation filter [66].

Body velocity $O_v \in \mathbb{R}^3$ and $O_\omega \in \mathbb{R}^3$: Two vectors, which describe the linear velocity $[ms^{-1}]$ and angular velocity $[rads^{-1}]$ of the robot's body. These can be obtained from a combination of an accelerometer and a gyroscope.

Body acceleration $O_a \in \mathbb{R}^3$ and $O_\alpha \in \mathbb{R}^3$: Two vectors, which describe the linear acceleration $[ms^{-2}]$ and angular acceleration $[rads^{-2}]$ of the robot's body. These can be obtained from a combination of an accelerometer and a gyroscope.

Straight line deviation $O_y \in \mathbb{R}$: The deviation in $[m]$ from a target straight line. This can be estimated via a simultaneous localisation and mapping (SLAM)

algorithm [67] and provides feedback to the robot to correct for deviations from a desired trajectory.

Foot touches $O_g \in \mathbb{R}^6$: A binary vector which indicates whether the tip of a leg is in touch with the ground. This can be measured with a mechanical switch on each leg.

Joint angle positions $O_\theta \in \mathbb{R}^{18}$: A vector, which contains the deviation of each joint from its neutral position in $[rad]$.

Joint angle velocities $O_{\dot{\theta}} \in \mathbb{R}^{18}$: A vector, which contains the angular velocity $[rads^{-1}]$ of each actuator about the joint axis.

Applied joint torques $O_\tau \in \mathbb{R}^{18}$: A vector, which contains the torque $[Nm]$ that is applied in order to rotate the joint to the desired joint angle.

External joint forces $O_F \in \mathbb{R}^{6\times6}$: A matrix which contains measurements of external forces and torques that are applied to each leg of the robot. Whenever the end-effectors (the tips of each leg) touch the ground or an obstacle, torques and forces are generated at the contact points. These "propagate" through the links of the robot. To keep the robot joints at their desired positions, counter-acting torques must be applied about each joint axis. The remaining components of the force and torque vectors are resisted by the structure of the robot [68]; these are called reaction forces/torques and can be measured with a 6 DoF force-torque sensor. To understand where to place these sensors, it is helpful to look at Equations (4.1) and (4.2) and Figure 4.2 which describe the static force/torque "propagation" [68] between links:

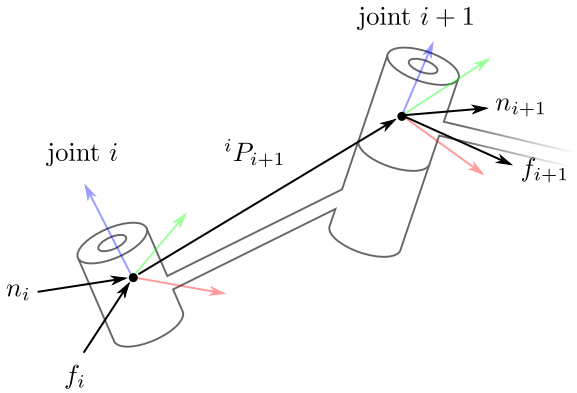


Figure 4.2: Static force/torque "propagation" through a link. (Adapted from [68])

$$ {}^if_i = {}^i_{i+1}R\,{}^if_{i+1} \tag{4.1} $$

$$ {}^in_i = {}^i_{i+1}R\,{}^in_{i+1} + {}^iP_{i+1} \times {}^if_i \tag{4.2} $$

Where ${}^if_i$ and ${}^in_i$ are, respectively, the forces and torques exerted on link i by link $i-1$, measured in the frame of link i; ${}^i_{i+1}R$ and ${}^iP_{i+1}$ are, respectively, the rotation and translation matrices that describe the transition from frame i to frame $i+1$. It becomes apparent that the external force/torque vectors observed at any joint are a linear transformation of the external force/torque vectors at the end-effector and thus it is sufficient to only put a sensor on one joint for each leg. We chose to put these on the body joints with servos $s^i_1,\ i \in [1, ..., 6]$.

Local heightmap (optional) $O_H \in \mathbb{R}^{32\times32}$: The CNN policies have an additional input, which is a 4 m^2 rectangular heightmap (see Figure 4.3) of the terrain around

the current position of the robot. This can be estimated using a depth-camera and the SLAM algorithm [67] and provides direct information about the terrain which could help in planning.

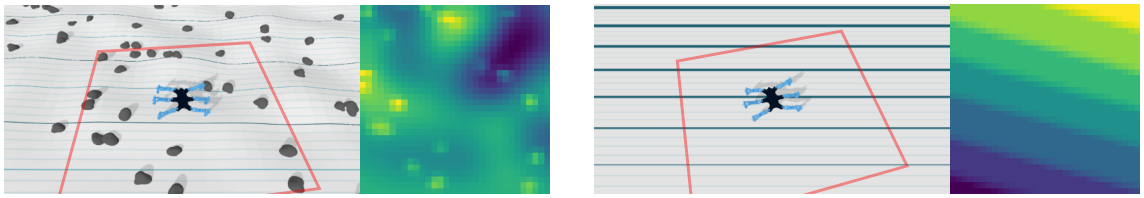


Figure 4.3: A visualisation of the heightmap that the robot "sees" on a terrain with bumps (left) and with stairs (right). The red rectangle is the range of the heightmap.

4.3.2 Observation Normalisation

In supervised learning, features with greater numeric values appear more important, and dominate over features with smaller numeric values [69]. This could bias the model into ignoring potentially informative features. This problem becomes detrimental when trying to find good policies in reinforcement learning [57]. We take two measures to ensure that all components in the observation space are in the same range.

First, before training we scale all bounded inputs with a known range to $[-1, 1]$. The joint angle measurements are normalised based on the values from §4.1. The maximum achievable angular velocity for a servo is 7 $rads^{-1}$ and the maximum torque that it can apply is 3.1 Nm [64] – these values are used to normalise the joint velocities and applied torques. For any unbounded inputs, such as *body velocity*, *body acceleration*, and *external joint forces*, we found an empirical mean and a standard deviation by running the reference gait; we then subtract this mean and divide by the variance for any new data.

Second, during training we keep track of the empirical mean O_μ and standard deviation O_σ of all observations that have been encountered. To normalise the observations, we subtract the mean O_μ, and divide by the standard deviation O_σ. As a result all observation components appear as if sampled from a distribution with a zero mean and unit variance.

4.4 Action Space

The output of a policy generates control signals that drive the actuators of the robot. The main methods for controlling robotic actuators are: setting the target applied torques [10, 37, 70], target joint velocities [71, 72], or target joint angles for proportional-derivative (PD) controllers [4, 14, 16, 40]. Policies that control the target joint angles for PD controllers have been shown to converge faster and result in more robust gaits, which are resilient to perturbations [73]. Thus, the 18-dimensional action space in our experiments controls the target angle on each joint of the robot.

One issue with the stochastic policies used in policy gradient reinforcement learning is that they can produce very different actions in two consecutive timesteps if

the policy's variance is large. This can be seen as zero-mean Gaussian noise added to the produced actions. This is tolerable when the robot is controlled via target joint velocities or torques since the integration of these quantities with respect to time acts as a low-pass filter. However, when the robot is controlled with target joint angles, the result of the stochastic policy is a gait in which the legs of the robot vibrate. Solutions to this instability include discretisation of the action space [16] or filtering the action signals [9, 12]. In these experiments, the action signals are filtered using a Butterworth filter [74] due to its maximally flat amplitude response and tolerable phase response. A 3$^{\text{rd}}$ order filter with a cut-off frequency of 5 Hz is used in all experiments. The filter's parameters were chosen empirically, after recording the mean frequency response of all actuators when using the reference controller from §4.2 (see Figures 4.4 and 4.5).

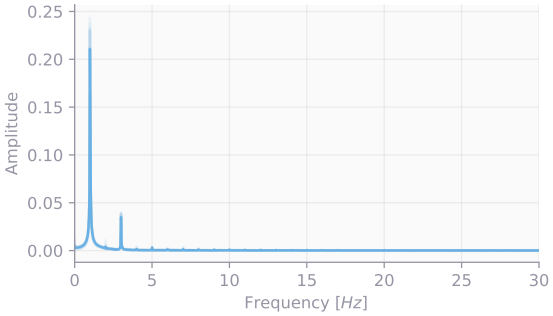


Figure 4.4: The mean frequency response from all joint-angle signals of the reference controller.

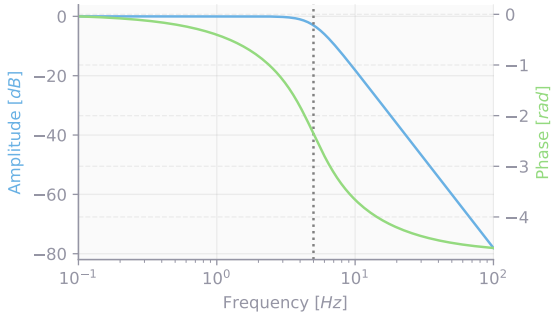


Figure 4.5: Frequency response of a digital 3$^{\text{rd}}$ order Butterworth filter with a cut-off frequency of 5 Hz.

4.5 Reward

The reward function specifies what the agent is trying to achieve. In this experiment, the goal of the agent is to walk forward along a given direction. Although a reward that is proportional to the forward velocity of the robot's body will yield some solution, we found that the resulting gait would likely look erratic. Through multiple trial-and-error, we added several other components to the reward function that yield a gait, which looks more natural while still achieving a high velocity. The weighted sum of these components is shown in Equation 4.3. It is important to note that a reward component with a negative weight will act as a penalty. The rest of this section will introduce the reward components and the reasoning behind them.

$$R = \omega_{vel}R_{vel} + \omega_{angle}R_{angle} + \omega_{line}R_{line} + \omega_{s2s3}R_{s2s3} + \omega_{coll}R_{coll} + \omega_{tor}R_{tor} \quad (4.3)$$

The *forward velocity reward* R_{vel} is a projection of the body frame forward linear velocity of the robot onto the y-axis of the world. This reward is maximised when the robot's body is aligned with the y-axis of the world and moves forward as quickly as possible. If θ_{yaw} is the orientation of the robot with respect to the world's z-axis in radians and O_v^0 is the x-component from the robot's linear velocity observation, the forward velocity reward is formally defined as:

$$R_{vel} = O_v^0 cos(\theta_{yaw}) \quad (4.4)$$

The *joint angle cost* R_{angle} is a function of the joint angles of the robot that penalises the agent if the actuators are close to their limits. This prevents the occurrence of a gait in which a joint is "stuck" at its limit. It is defined as the mean of the tenth power of the joint angles (Equation (4.5)). Raising the joint angle values – which are in the range $[-1, 1]$ – to the tenth power means that this cost is only significant when the actuators are at their extremes.

$$R_{angle} = \frac{1}{18} \sum_{i=1}^{18} {O_\theta^i}^{10} \tag{4.5}$$

The *straight line cost* R_{line} penalises the robot for deviating from the middle of a terrain. It is defined as the absolute value of the robot's y-position in the world's frame (Equation (4.6)). As a result, it is minimised when the robot stays on the y-axis.

$$R_{line} = |O_y| \tag{4.6}$$

The s_2 - s_3 *difference cost* R_{s2s3} penalises the agent if the difference between the joint angles on servos s_2 and s_3 on each leg is high. The minimisation of this cost happens when the joint angles on these servo pairs are the same, which means that the legs are perpendicular to the ground. As a result, the contact area with the ground is greater and this allows the robot to walk on slippery terrains or inclines. It is defined as follows:

$$R_{s2s3} = \frac{1}{6} \sum_{i=1}^{6} \left| O_\theta^{i,2} - O_\theta^{i,3} \right| \tag{4.7}$$

The *self-collision cost* R_{coll} has a binary output, which is 1 if any of the robot's legs touch and 0 otherwise. It encourages the robot to prevent self-collisions which can be detrimental on a physical robot.

The *applied torque cost* R_{tor} penalises the agent for applying torque to the actuators. This cost is minimised when the robot is stationary. However, its combination with the forward velocity reward encourages the robot to walk forward while preventing aggressive motions. It is defined as:

$$R_{tor} = \frac{1}{18} \sum_{i=1}^{18} {O_\tau^i}^2 \tag{4.8}$$

The weights of these reward components are crucial to the quality of the learnt gait. They have been inspired from previous work [10, 39] and further tuned empirically to the values in Table 4.1. The criteria for tuning the weights were: speed of convergence, smoothness of the learnt gait, occurrence of stuck joints, occurrence of self-collisions, forward velocity of the learnt gait, and ability to stay on the target line without much deviation.

Weight	ω_{vel}	ω_{angle}	ω_{line}	ω_{s2s3}	ω_{coll}	ω_{tor}
Value	1.0	-1.0	-0.4	-0.3	-0.2	-0.5

Table 4.1: Weight values for the components of the reward function.

4.6 Terrains

The main objective of the project is to test the adaptability of the agent to environmental changes. In this section, we introduce methods for procedurally[2] generating parameterised terrains. The parameters affect the structure of the generated terrains and are used to control their difficulty. We present structured terrains, which have a predictable pattern, and unstructured terrains, which are pseudo-random and have no predictable structure.

4.6.1 Terrain with Stairs

Stairs are an example of a structured terrain. They are a common obstacle to robots which operate in human-made environments. The stair terrain in this work is defined by three parameters: *stair depth*, *stair height*, and *number of stairs*. Figure 4.6 shows examples of generated stairs for a range of stair heights.

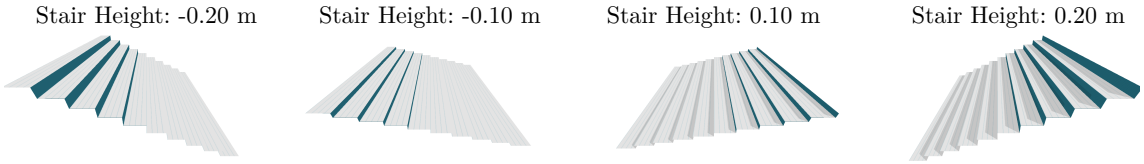


Figure 4.6: Examples of generated stairs for a range of stair heights. The stair depth and the number of stairs are fixed at 0.4 m and 10 respectively.

4.6.2 Terrain with Bumps

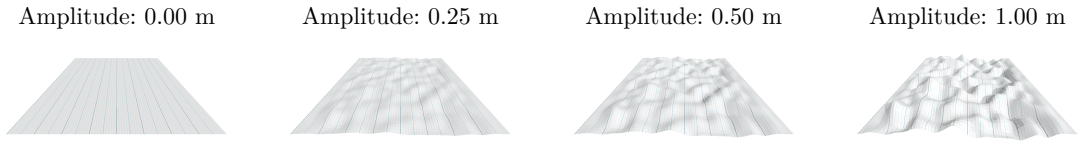
Although most of the environments encountered by humans are structured (e.g. stairs, roads, etc.), a robot is likely to operate in unstructured terrains (e.g. in a forest, or on a field). Thus, a method for generating randomised unstructured terrains is required that still allows for parameterised control. A simple mapping from coordinates in the xy-plane to random z-coordinates would generate an unrealistic terrain, since there will be no correlation between the height of adjacent xy-coordinates. Perlin noise [75, 76] (see appendix A.1) can generate patterns which resemble those of wild terrains and was used to generate the bumpy terrains (see Figure 4.7).

Each bumpy terrain is parameterised by: the *amplitude* which defines the maximum height of a bump in meters; the *frequency* which defines the average number of bumps per meter; and the *elevation* which defines whether the terrain goes up or down in meters. The bumps smoothly disappear at the edges of the terrain, which allows for the combination of multiple terrain types.

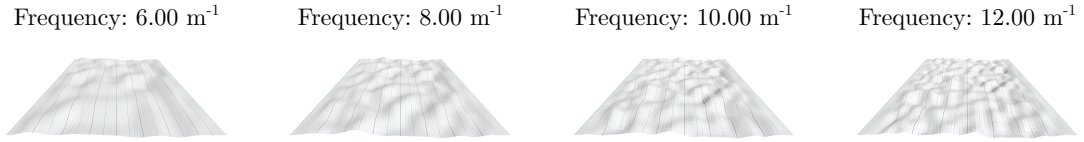
4.6.3 Additive Stones

In addition to the terrains above, we created terrain enhancements which can be added on top of any terrain type (see Figure 4.9). Stones can be encountered nearly anywhere in the real world; in simulation, they are used to add small-scale randomness to both structured and unstructured terrains. We created a stone mesh

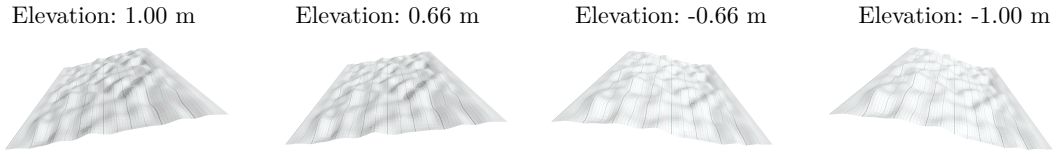
[2]Procedurally means that the terrains are generated algorithmically rather than manually.



(a) *Frequency* $= 10.00\ m^{-1}$; *Elevation* $= 0.00\ m$.



(b) *Amplitude* $= 0.50\ m$; *Elevation* $= 0.00\ m$



(c) *Amplitude* $= 0.50\ m$; *Frequency* $= 10.00\ m^{-1}$

Figure 4.7: Examples of generated bumps for a range of amplitude, frequency, and elevation values.

by distorting a unit sphere with Perlin noise [75, 76] (see Figure 4.8). Instances of this mesh are then scattered across a terrain by sampling the x and y coordinates, and the orientation from a uniform distribution. The z-coordinates are set based on the heightmap value of the terrain at [x, y]. The additive stones are static on the terrain and are parameterised by: the *stone size*, which defines the scale of the unit sphere which generates the stone mesh; and the *stone density* which defines the average number of stones that cover the terrain per square meter.

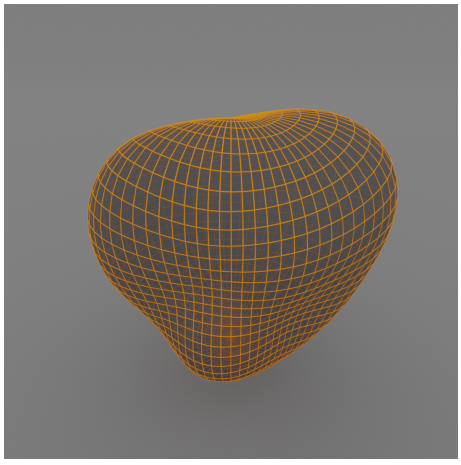
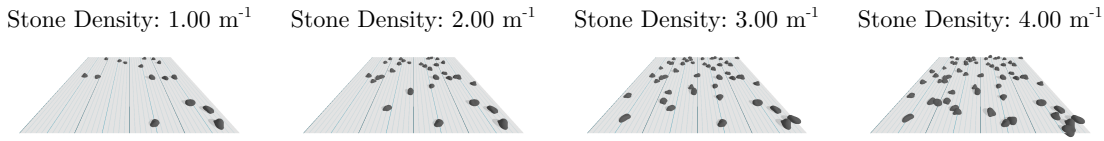


Figure 4.8: The mesh of a stone – a unit sphere distorted by Perlin noise.



(a) *Stone Size* $= 10.0cm$

(b) *Stone Density* $= 2.0m^{-1}$

Figure 4.9: Examples of generated stones for a range of stone density and stone size values.

4.7 Randomisations

This section includes a detailed description of all randomised parameters during training. The ADR hyperparameters used in the experiments are summarised in Table A.2.

The robot initialisation is randomised by two parameters – the initial *robot y* position and the initial *robot yaw* orientation. The first benefit these randomisations provide is that they decorrelate the trajectories sampled from the parallel workers, which significantly speeds up the beginning of the learning process. The second benefit is that they allow the robot to learn how to recover from an arbitrary deviation from its desired location and orientation.

The terrain structure is randomised by four parameters – stair terrains are defined by the *stair height* while bumpy terrains are defined by the *amplitude* of the bumps and the *elevation gain*. The *stone density* is applicable to both terrains. We chose not to randomise the *stair depth*, *frequency* and *stone size* parameters of these terrains since: including these parameters does not contribute much to the control over difficulty; and the terrain difficulty does not always increase with these parameters (see chapter 6 for discussion). The terrain type – stairs or bumps – is chosen with an equal probability, except when the boundary of a parameter that describes one of these terrains is sampled; in that case the terrain that is described by the parameter is chosen. The *gravity* and *friction* parameters of the simulator are also randomised.

Lastly, we add random noise to the observation space and randomly reduce the available power to several actuators. Since it is impossible to predict everything that could be encountered by a physical robot, these two randomisation aim to simulate the effect of a generic defect of a physical robot. The observation noise in Equation (4.9) is added to the observations and consists of two components: n_0 is sampled once every episode and it acts as a constant bias that is added to all sensors; n_1 is sampled on each timestep and acts as generic electrical noise. The power scale vector in Equation (4.10) is sampled once each episode and is used to scale the maximum available torque of each servo. The softmax with temperature in that function creates a sparse vector in which most values are zero, apart from a few which are in $[0, 1]$. The effect of this is that in each episode a low number of

actuators will be less effective than usual.

$$O_{noise} = n_0 + n_1 \text{ for } n_0 \sim \mathcal{N}(0, |\lambda_i|), n_1 \sim \mathcal{N}(0, |\lambda_j|) \tag{4.9}$$

$$P_{scale} = (1 - \lambda_k \cdot \text{softmax}(10 \cdot \boldsymbol{n}_2))^4 \text{ for } \boldsymbol{n}_2 \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \tag{4.10}$$

Chapter 5

Results

This chapter empirically quantifies the effect of ADR on the three policy types – FC, LSTM, and CNN – and the performance of each on in-distribution environments and out-of-distribution damages. The end of this chapter contains qualitative analysis of the produced locomotion gaits.

5.1 Training

We start by training each policy architecture on a flat terrain without ADR[1] for 4 million timesteps[2] which results in three agents – *FC*, *LSTM*, and *CNN*. This allows for a comparison between agents trained with and without ADR, but also provides model parameters which are used to initialise the later models. Figure 5.1 shows the training progress of these agents. On the flat terrain environment all agents converge to similar values.

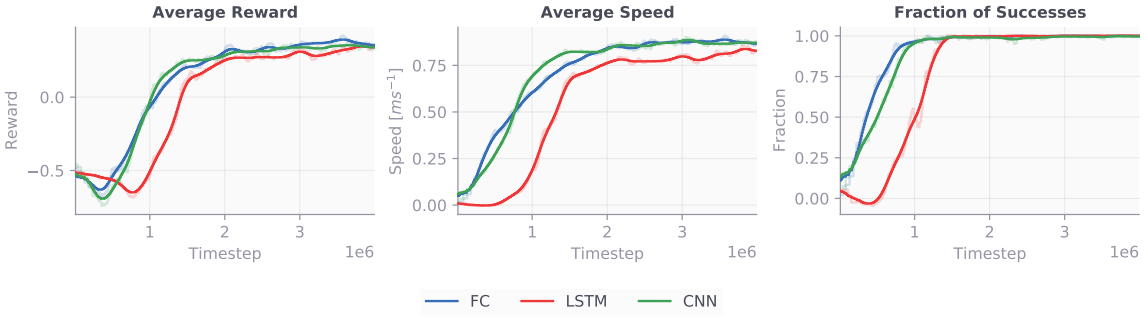


Figure 5.1: Training progress of the *FC*, *LSTM* and *CNN* agents without ADR. All agents converge on a similar average reward per timestep of 0.3 (left). The average speed with which they traverse the terrain from start to finish (middle) is 0.8 m/s. At the end of training all agents manage to traverse the terrain on every attempt (right).

The model parameters from the agents above are then used to initialise three new agents – *FC + ADR*, *LSTM + ADR*, and *CNN + ADR* – which are trained with ADR for 30 million timesteps. Figure 5.2 shows the training progress of these

[1] Although ADR is not used here, the robot y and robot yaw are still sampled from a small distribution to decorrelate the parallel sampling process in the beginning of training.

[2] One timestep is a single interaction step between the agent and the environment.

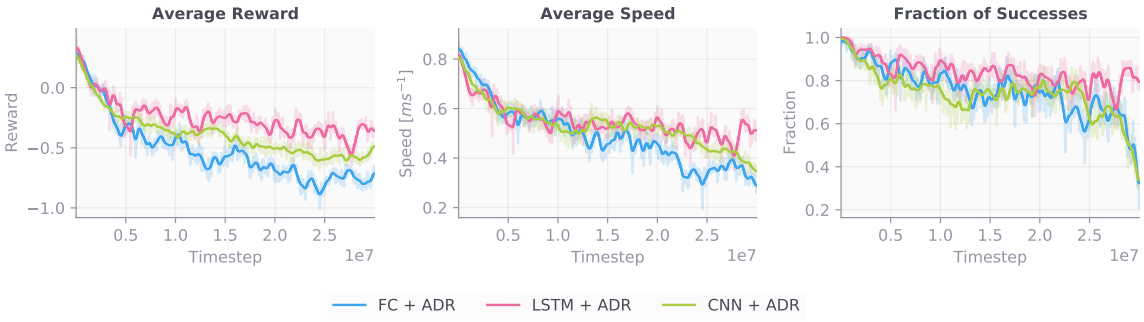


Figure 5.2: Training progress of the *FC + ADR*, *LSTM + ADR* and *CNN + ADR* agents. All agents, apart from the *LSTM* one, fail to maintain a reasonable performance at the end of training and the fraction of successful traversals of the terrain approaches 0.

agents. The average reward, average speed and fraction of successes for each agent drop over time – this is expected since the environments become progressively harder. However, ADR seems to be detrimental to the *FC + ADR* and *CNN + ADR* agents and their success rates sharply drop after about 27 million timesteps. The *LSTM + ADR* agent, however, manages to keep up with the harder environments.

This observation is reflected in the ADR entropy and the sizes of all distributions. Figure 5.3 shows the evolution of the ADR distributions and the entropy with training. In the case of the *FC + ADR* and *CNN + ADR* agents, all distributions collapse to a size of nearly zero and the ADR entropy approaches $-\infty$ at the end of

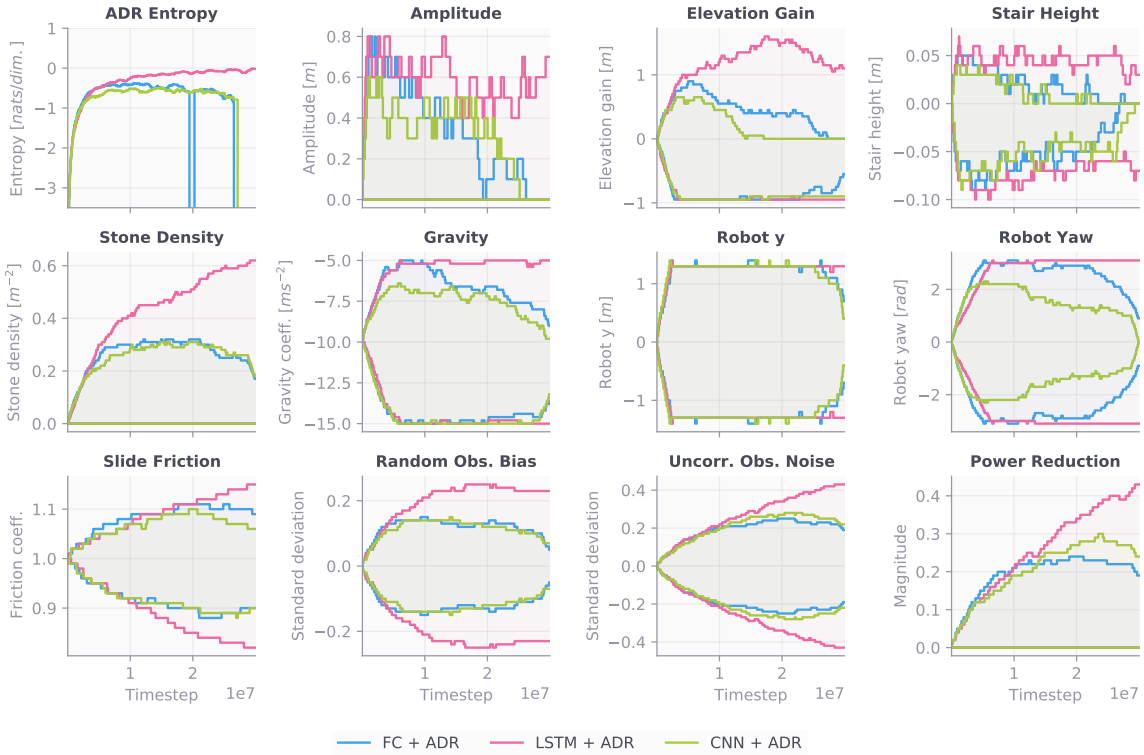


Figure 5.3: Expansion/shrinkage of the ADR parameter distributions and their corresponding entropy during the training of the *FC + ADR*, *LSTM + ADR* and *CNN + ADR* agents.

training. On the other hand, the ADR entropy and most distributions – with the exception of amplitude, elevation gain and stair height – continue to grow for the *LSTM + ADR* agent.

5.2 Adaptation to Environmental Randomisation

In the following experiment, the ADR algorithm is executed 5000 times for the reference agent and the six agents from the previous section. The boundary sampling probability p_B is set to 1, meaning that at each update step, one parameter has one of its boundaries evaluated. The purpose of this experiment is to test the ability of each agent to adapt to randomisations – an agent that adapts well will have a high ADR entropy and large parameter distributions.

Figure 5.4 shows the expansions of all distributions for the seven agents. Since the policies of the *FC + ADR* and *CNN + ADR* were destroyed by the ADR algorithm, the environments of these agents never expand and the ADR entropy is $-\infty$. Apart from the low boundary of the *stair height* parameter which is expanded the most by the *Reference* agent, all parameter boundaries are expanded the most by the *LSTM* and *LSTM + ADR* agents.

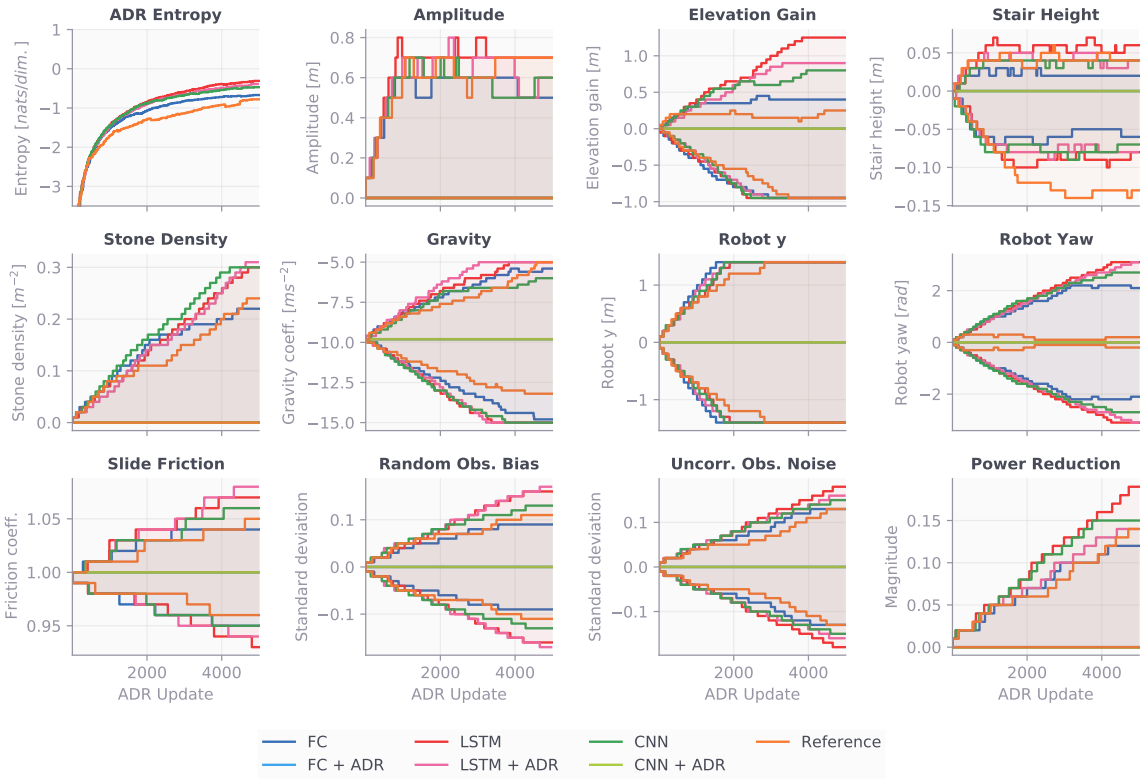


Figure 5.4: Expansion/shrinkage of the ADR parameter distributions and their corresponding entropy during the evaluation of all agents.

Figure 5.5 is a parallel coordinates plot, which shows the size of each parameter distribution after 5000 ADR updates. Although, the two *CNN* agents have a way of "seeing" the terrain, they fail to expand parameter distributions related to the terrain's structure more than the two *LSTM* agents. Even though the *LSTM + ADR* agent has been trained on a similar distribution over environments that was

used in this evaluation, in many cases it does not manage to increment past the distribution sizes than the *LSTM* agent manages to solve.

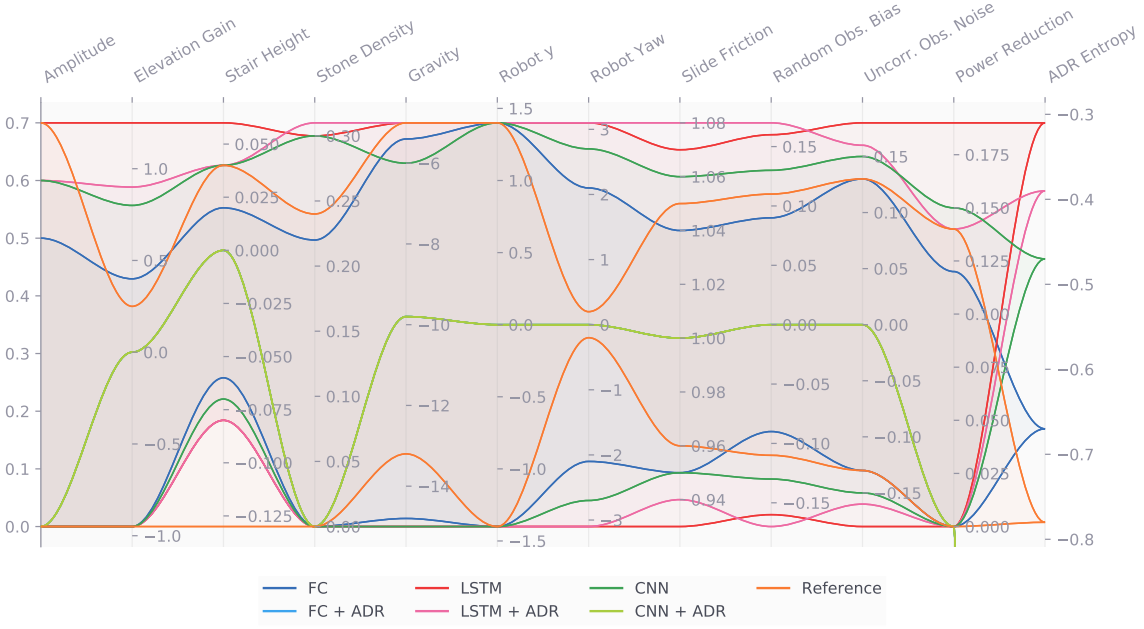


Figure 5.5: The final size of all parameter distributions after 5000 ADR updates for the 7 agents. Since the policies of the *FC + ADR* and *CNN + ADR* agents were destroyed by the ADR training, their corresponding distributions have a size of 0.

5.3 Performance Within the ADR Distributions

Although the ADR entropy and the individual parameter boundaries are representative of the amount and type of randomisation each agent can handle, these measures do not indicate the performance of the agents on different parameter combinations

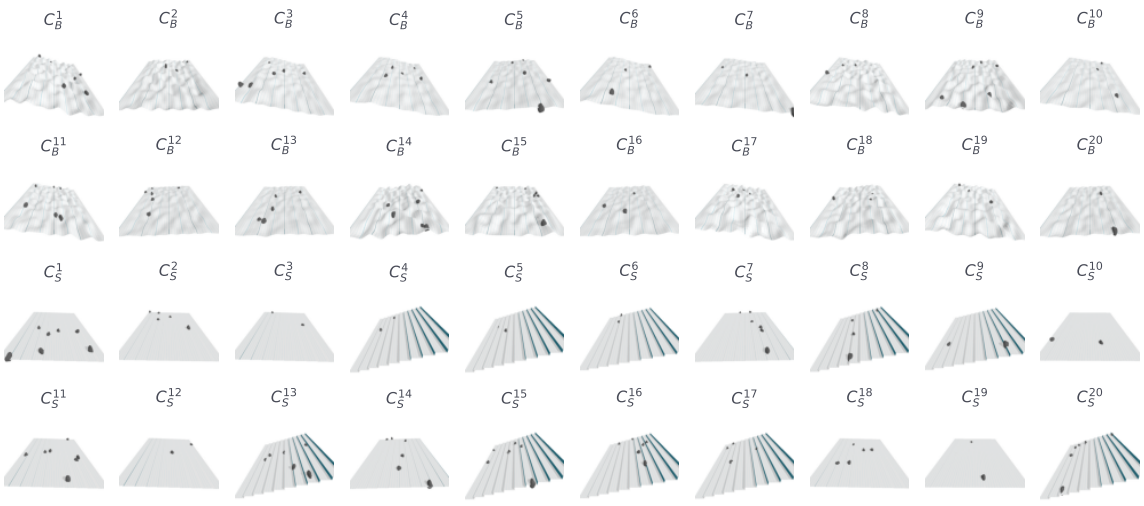


Figure 5.6: Visualisation of terrains in each cluster. Full cluster definition is available in Tables A.5 and A.6.

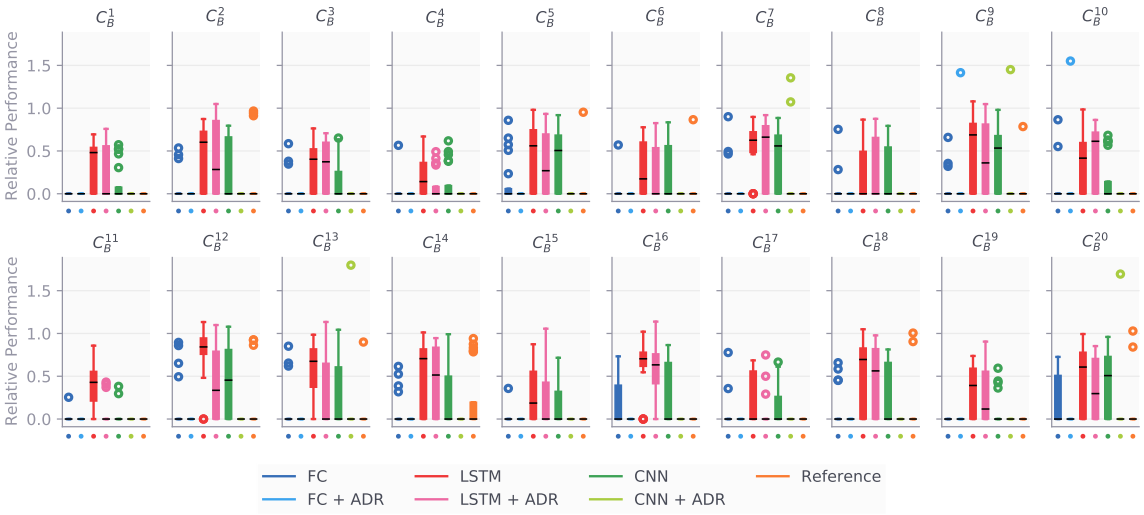


Figure 5.7: The relative performance of all agents on bumpy terrain environments that are sampled from the discretised ADR parameter space. The cluster centres $C_B^i,\ i \in [1, ..., 20]$ represent a particular ADR parameter combination – these are defined in Table A.5.

sampled from within the ADR distributions. Measuring the performance by sampling these distributions on a uniform grid would be impractical since the number of grid points grows exponentially with the number of ADR parameters. An alternative method is to discretise the ADR parameter space into a given number of clusters using a centroidal Voronoi tessellation (CVT) [21]. In this experiment, the *random k-means method* [77] was used to determine 20 cluster centres inside the ADR distributions. The boundaries of these distributions were set to the empirical maximum that any agent was able to solve at any point during training or evaluation (boundary values are available in Table A.3). Then, for each agent the performance within each cluster was recorded – this was achieved by sampling 20 ADR parameter

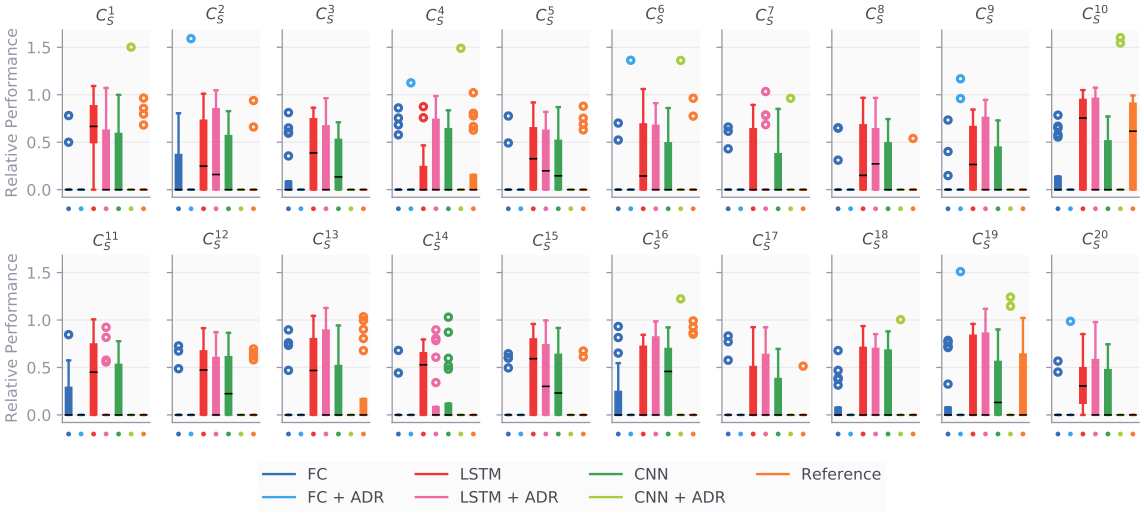
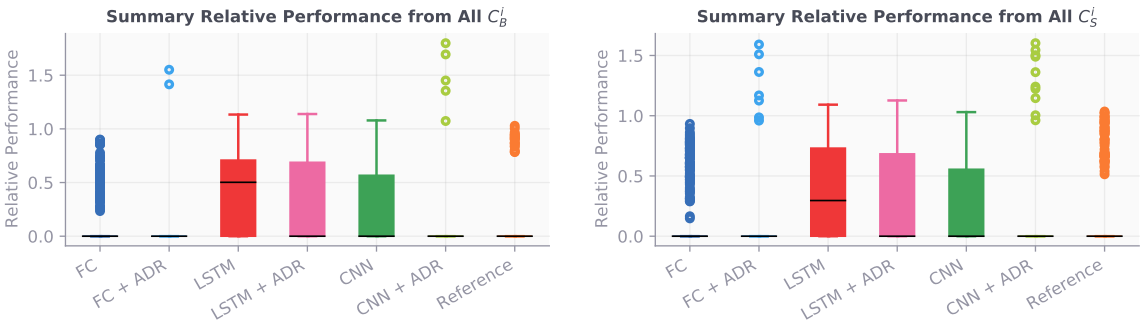


Figure 5.8: The relative performance of all agents on stair terrain environments that are sampled from the discretised ADR parameter space. The cluster centres $C_S^i,\ i \in [1, ..., 20]$ represent a particular ADR parameter combination – these are defined in Table A.6.



(a) Summary on terrains with bumps.

(b) Summary on terrains with stairs.

Figure 5.9: A summary of the relative performance of all agents in the whole ADR parameter space.

combinations with a uniform probability from each cluster and running the agent.

Since the *amplitude* and *elevation gain* parameters have no effect in terrains with stairs and the *stair height* parameter has no effect in terrains with bumps, a separate evaluation was done for the terrains with bumps and stairs – only the relevant parameters were included for each terrain type. Figures 5.7 and 5.8 show the relative performance of all agents within each cluster of the discretised ADR parameter space, while Figures 5.9a and 5.9b show an overall summary of the relative performance. Figure 5.6 is a visualisation of terrains from each cluster for the parameters that can be visualised (such as *amplitude*, *stair height*, etc.). The relative performance is defined as the ratio between the expected forward velocity achieved during evaluation and the expected forward velocity of the undamaged robot on the simplest flat terrain. The use of this metric allows for an obvious way to determine how much worse/better the robot is performing in a particular setting than usual. It also allows us to compare how well the different agents adapt, regardless of their initial speed.

First, the results on the bumpy terrain are analysed (Figures 5.7 and 5.9a). Despite several outliers, the *FC*, *FC + ADR*, *CNN + ADR*, and *Reference* agents have a zero median relative performance in all CVT clusters. This means that, in general, they failed to traverse the full terrain. The *LSTM + ADR* and *CNN* agents managed to solve some of the environments, but their overall median relative performance is still zero. Only the *LSTM* agent achieved non-zero relative performance in all clusters (apart from C_B^8 and C_B^{17}) – in general, it managed to complete the environments with 0.5 times its normal forward velocity.

Next, the results on the stairs terrain are analysed (Figures 5.8 and 5.9b). The *FC* and *CNN* agents successfully solve environments in several CVT clusters, but their median relative performance is still 0. The *ADR* versions of these agents fail to solve any of these problems. The *Reference* agent performs well in the cluster around centre C_S^{10}, but fails to complete most other environments. Again, the *LSTM* agent solves the environments in most clusters with a median relative performance of 0.3 – its *ADR* version comes second.

Overall, all agents apart from the *LSTM* agent fail to achieve a satisfactory result. A comparison between the normal and *ADR* versions of the *FC*, *LSTM* and *CNN* agents reveals that *ADR* was detrimental to the ability of these agents to solve diverse environments.

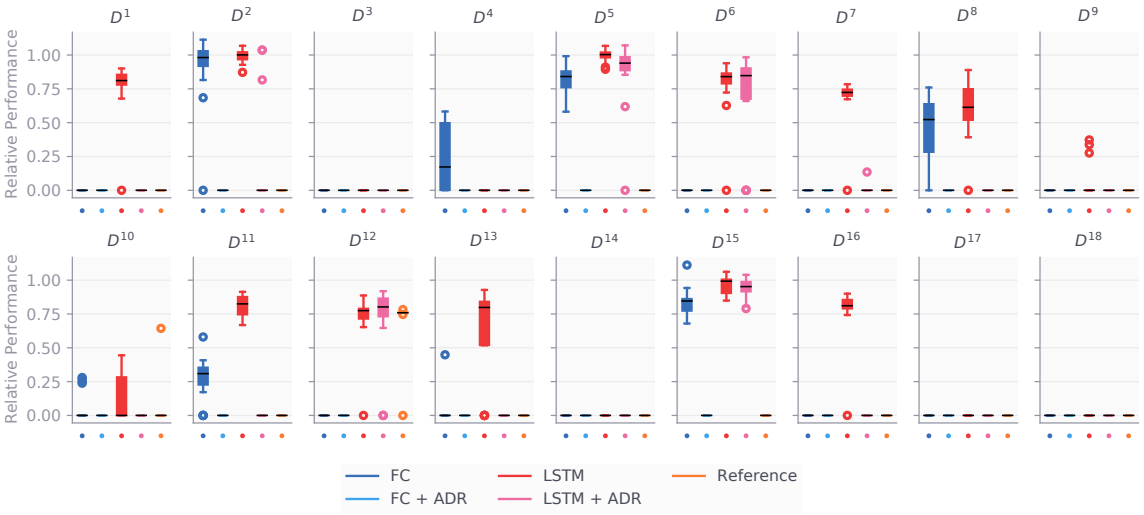


Figure 5.10: The relative performance of all agents while 18 different damages are in effect. The effects of all damages D^i, $i \in [1, ..., 18]$ are defined in Table A.4.

5.4 Performance on Unseen Damages

The previous section compared different agents on a common set of environments which have been encountered during training. Although the variety in these environments attempts to capture a wide range of situations, it is impossible to predict every situation the robot might encounter after deployment. In this section, we test the ability of each agent to adapt to 18 out-of-distribution damages on flat terrains. These include different combinations of a missing leg, two missing legs, and legs that have lost power and are free-hanging. Figures 5.10 and 5.11 summarise the results after measuring the relative performance of each agent with each damage in 20 repetitions. Once again, the *LSTM* agent leads in terms of relative performance and successfully adapts to 12 of the damages. The *FC* agent adapts to 6 of the damages, but with lower relative performance to the *LSTM* agent. The *ADR* versions of these agents perform worse in general. The *Reference* agent manages to adapt to a single damage. Both *CNN* agents are not included in this experiment, since their policy architecture acts identically to that of the *FC* agents on flat terrains. Although the *LSTM + ADR* agent has been trained in environments with different friction and gravity coefficients, random power reduction of the actuators and noise added to the sensors, its performance is mostly surpassed by the *LSTM* agent.

Overall, the results are similar to the previous section – the *LSTM* agent per-

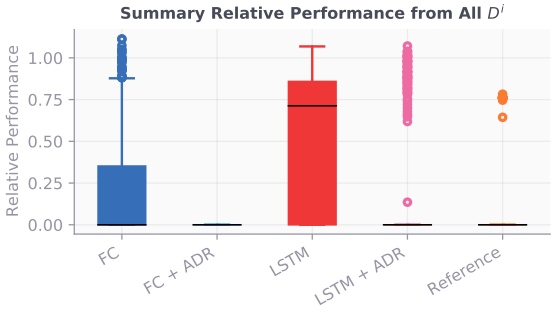


Figure 5.11: A summary of the relative performance of all agents on all damages.

forms best, with a median relative performance of 0.7. ADR is, again, detrimental to the quality of the agents, and none of the other agents achieved a non-zero median relative performance.

5.5 Gait Analysis and Qualitative Results

Lastly, the dynamics and the quality of the learnt gaits are analysed. A measure that reflects how dynamic a gait is is the body velocity to joint angle frequency ratio. Moving fast purely due to the fast motion of the joints is inefficient and could even damage them. A dynamic and fast gait would keep a moderate joint angle frequency and instead use "hopping" to increase the speed of the robot. As a result, a more dynamic gait would have a higher body velocity to joint angle frequency ratio. Figure 5.12 shows this metric for all agents on a flat terrain and an undamaged robot. The *FC* and *CNN* agents have the most dynamic gaits, although the higher variance and the outliers hint that they are unstable. The *LSTM* agent comes slightly behind the *FC* and *CNN* agents.

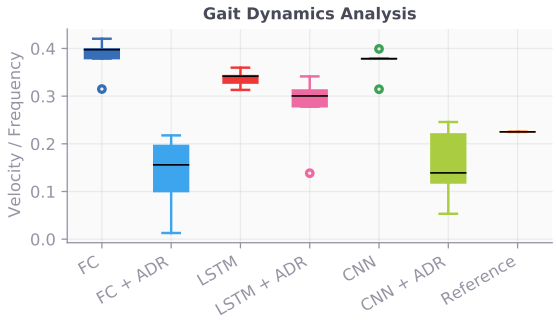


Figure 5.12: Gait dynamics analysis of all agents by comparing their body velocity to joint angle frequency ratios.

We also provide supplementary representative videos[3] of the agents. A visual inspection reveals that the gaits are indeed dynamic. The *LSTM* agent adapts to damages in less than a second and walks straight, while the reference agent walks in circles. When an obstacle is encountered by the *LSTM* agent, it immediately corrects for the error in orientation. A problematic situation for all agents is going upstairs due to the lack of visual feedback.

[3]Supplementary video playlist here: youtube.com/playlist?list=PLi_LG60WoGxP3rVd6e_SIhRHBHSyI_cGP

Chapter 6

Discussion

In this investigation, we assessed the significance of different policy architectures and automatic domain randomisation on the ability of robots to adapt to a variety of terrains and damages. With methods based on proximal policy optimisation, we compare seven different agents on structured and unstructured terrains and out-of-distribution damages. The agents we test are based on feedforward, recurrent, and convolutional neural networks – an agent based on each network is evaluated when trained with and without automatic domain randomisation. A seventh agent found by the intelligent trial and error algorithm is evaluated as a reference. The ability to adapt to terrains is tested on two parameterised environments consisting of stairs or bumps with 11 parameters that control the difficulty. The ability to adapt to unseen damages is tested on 18 damages which include a missing leg, an unpowered leg, and two missing legs. Our investigation provides several insights into what matters in robotic adaptation.

Policies with memory: In all experiments, a memory-augmented LSTM policy network achieved the highest performance and ability to adapt to randomisations. We believe that the reason for this is the ability of the agent to remember past observations and reason about the state in a partially observable MDPs. For instance, suppose that a robot is damaged and one of its joints is stuck in the neutral position. A policy without memory would observe the state and at some instances of time, having that joint at its neutral position would appear normal. A policy with memory, however, would detect that the joint is not moving at all. This is particularly highlighted when comparing the *FC* and *LSTM* agents. The *FC* agent performs as well as the *LSTM* agent on a flat terrain, where the observation space describes the environment sufficiently well. However, the *FC* agent fails at many of the adaptation tests, while the *LSTM* performs significantly better. Our findings align with the hypothesis that memory in the agent is an important ingredient to meta-learning and it matches previous observations in robotic manipulation problems [16, 50]. Unlike map-based adaptation techniques [4–9], our method achieved true non-episodic adaptation in 12 out of 18 damages.

Meta-learning and domain randomisation: Although our results suggest that domain randomisation is not helpful in adaptation and meta-learning, we do not believe that the relationship is causal. In fact, previous work has reached the opposite conclusion [5, 16, 48–50]. In our setup, neural networks with direct mapping from observations to actions were used. Although the size of these policy networks is comparable to previous work in the field, the low inductive bias in our policies

prevented the agent to learn. Whereas we provide the full observation vector to a neural network that generates actions, [14, 16, 39] take a hierarchical approach where one network generates a target plan (e.g. target foot positions) whilst a second network tries to achieve this target plan. This way of explicitly adding domain knowledge to the policy architecture significantly simplifies the problem, making it easier to learn. Further, it creates subgoals that instruct the agent how to behave in the short-term as opposed to our problem formulation which mainly encourages the robot to maximise the long-term goal of walking forward fast.

Observing a heightmap of the terrain: The addition of a terrain heightmap, processed through a CNN, to the agent also caused inferior performance. Following the same reasoning from the previous paragraph, we believe that the problem is caused by a low inductive bias in the policy architecture.

Limitations of ADR: A limitation of ADR is that it models the distribution over environments as a uniform distribution and only evaluates it at its boundaries for expansion. This evaluation might not always be representative of the agent's performance overall. Suppose that the agent is evaluated on an environment with obstacles and the environment is parameterised by the size and the density of these obstacles (e.g. see §4.6.3). If the density is small, the size can grow unreasonably since it will be unlikely to encounter an obstacle during an episode. A large performance data buffer would reduce the impact of this issue, but it would also slow down the distribution update rate.

Chapter 7

Conclusion

We investigate the suitability of three reinforcement learning policy architectures and automatic domain randomisation to non-episodic environment and damage adaptation of robots. Seven agents are compared in our evaluations – *FC*, *LSTM* and *CNN* which are trained without ADR; *FC + ADR*, *LSTM + ADR* and *CNN + ADR* which are trained with ADR; and a *Reference* open-loop agent from [4]. These were trained in parameterised environments with a gradually increasing complexity. Our findings show that memory-augmented policies are crucial for adaptation and manage to adapt to a wide range of terrains and 12 out of 18 unseen damages whilst keeping a reasonable relative performance, greater than 0.5. Contrary to our expectations, ADR did not improve the ability to adapt and completely destroyed the *FC* and *CNN* agents, whilst significantly reducing the performance of the *LSTM* agent. Overall, our work provides an important insight into what matters in non-episodic adaptation and a partial solution to the problem. Further, it provides a modular codebase which will make further investigation easier.

There are many exciting directions for investigation. The effect of hierarchical policy architectures, combined with ADR, might be a way of harnessing the benefits of large domain variation in reasonable training time. A modification of the ADR distribution with a more expressive probability distribution might allow for even more variation in the domain and the discovery of niche parameter combinations. A possible option is to use a Gaussian mixture with update logic inspired by [78]. One of the difficulties we faced was the control over the gait frequency – its integration into the current learning procedure would be interesting. Further, a robust method for controlling different aspects of the learnt gait, such as speed, orientation, and body height from the ground, will significantly increase the applicability of this research to real scenarios. Lastly, it would be useful to know the contribution of each ADR parameter to the performance of the agent. A simple way to analyse this would be to measure the Pearson correlation coefficient between each parameter and the relative performance. Our current way of measuring relative performance does not allow this, since an agent that fails to complete a terrain gets a relative performance of zero. As a result, a new method for measuring performance is required.

Chapter 8

Project Management

At the beginning of the project, a project plan in the form of a Gantt chart was made (see Figure 8.1). Despite that the predicted time schedule was not followed strictly, this plan was extremely helpful in providing a big picture of everything that needs to be done in the beginning, and gave a sense of whether the project was ahead or behind schedule. It also served as a checklist that ensured everything is completed. In the short-term, we used the Scrum[1] framework for setting weekly goals. These goals were discussed weekly with my supervisor. Overall, all targets in the project plan were achieved. To prevent an over-ambitious project, or one that fails due to unforeseen circumstances, a pessimistic prediction on the required execution time was made. This, and the careful weekly planning allowed for extra objectives, such as a visual heightmap input and a CNN policy, to be included.

To prevent a catastrophic project failure due to loss of project files, every single feature in the codebase was immediately pushed to a remote repository on GitHub[2]. An extra copy was also kept on the Iridis 5 cluster, so there were 3 mirror copies of the project at all times.

A contingency plan was also made at the beginning of the project (see Table 8.1). This plan proved useful twice during the project as two of the risks occurred. First, the Iridis 5 cluster was occupied by high-priority users for an extensive amount of time during a period of hyperparameter optimisation. This task required quick evaluation cycles and to prevent a negative impact on the project, I wrote automated scripts that ran overnight on my laptop. Second, the initial RL framework we used[3] had an extremely poor and slow implementation of LSTM policies, which could significantly impact the project aims and prevent us from achieving some of the targets. Since this was a high impact risk, I temporarily stopped following my usual plan and rewrote the codebase to work with an alternative library. An event that was not anticipated during the initial plan of the project was the paternity leave of my supervisor. However, he made this clear well in advance and organised an introductory meeting with one of his PhD students. The weekly meetings with his student have been of extreme help and had a major influence on the results section of this dissertation. Although I do not think that this event impacted the project negatively at all, I do believe that the project would have taken a slightly different direction if it was conducted with my supervisor all the way through.

[1] https://www.scrum.org/resources/what-is-scrum
[2] https://github.com/
[3] https://stable-baselines.readthedocs.io

Risk	Likelihood	Impact	Action
Iridis 5 is down or busy	Low	Medium	Complete all computationally expensive tasks early. Run any tasks left on local machine. If urgent, use project budget to hire a virtual machine.
Bug in open-source library stops progress	Medium	High	Depending on the scale of the bug either write own implementation or search for an alternative library.
Hardware failure of storage devices	Low	High	Checkout the latest remote version of the project from GitHub.
Slowdown due to COVID-19	Medium	High	In case of a high severity, apply for a project extension.

Table 8.1: Contingency plan.

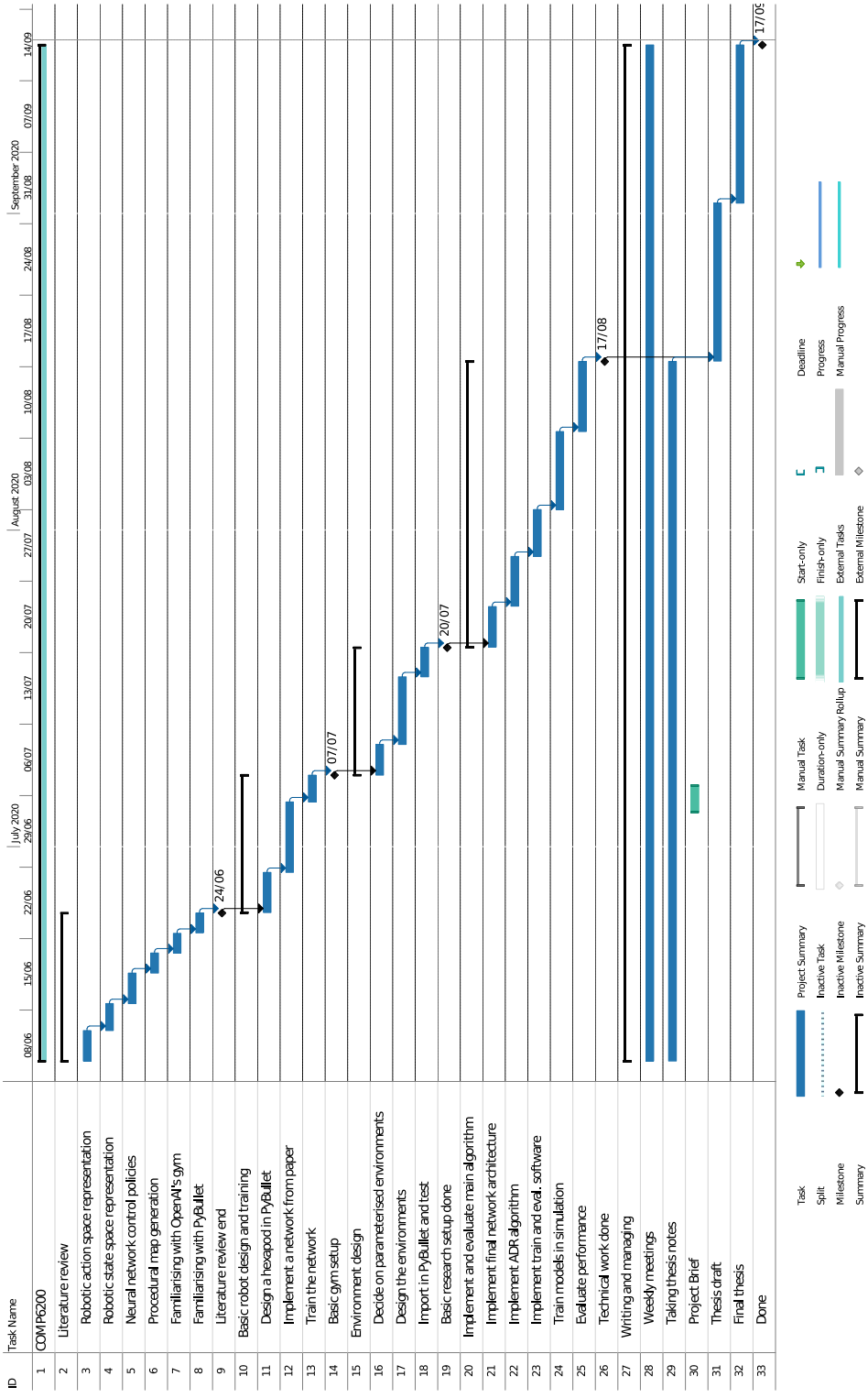


Figure 8.1: Initial project plan, made before the start of the project.

Bibliography

[1] B. Siciliano and O. Khatib, *Springer handbook of robotics*. Springer, 2016.

[2] K. Chatzilygeroudis, V. Vassiliades, F. Stulp, S. Calinon, and J.-B. Mouret, "A survey on policy search algorithms for learning robot controllers in a handful of trials", *IEEE Transactions on Robotics*, 2019.

[3] R. A. Brooks, "Intelligence without representation", *Artificial intelligence*, vol. 47, no. 1-3, pp. 139–159, 1991.

[4] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals", *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[5] D. M. Bossens and D. Tarapore, "QED: using Quality-Environment-Diversity to evolve resilient robot swarms", *arXiv preprint arXiv:2003.02341*, 2020.

[6] V. Vassiliades, K. Chatzilygeroudis, and J.-B. Mouret, "Using centroidal voronoi tessellations to scale up the multi-dimensional archive of phenotypic elites algorithm", *arXiv preprint arXiv:1610.05729*, 2016.

[7] K. Chatzilygeroudis, A. Cully, and J.-B. Mouret, "Towards semi-episodic learning for robot damage recovery", *arXiv preprint arXiv:1610.01407*, 2016.

[8] K. Chatzilygeroudis, V. Vassiliades, and J.-B. Mouret, "Reset-free trial-and-error learning for robot damage recovery", *Robotics and Autonomous Systems*, vol. 100, pp. 236–250, 2018.

[9] D. Tarapore and J.-B. Mouret, "Evolvability signatures of generative encodings: Beyond standard performance benchmarks", *Information Sciences*, vol. 313, pp. 43–61, 2015.

[10] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami, *et al.*, "Emergence of locomotion behaviours in rich environments", *arXiv preprint arXiv:1707.02286*, 2017.

[11] X. B. Peng, G. Berseth, and M. Van de Panne, "Terrain-adaptive locomotion skills using deep reinforcement learning", *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–12, 2016.

[12] X. B. Peng, E. Coumans, T. Zhang, T.-W. Lee, J. Tan, and S. Levine, "Learning agile robotic locomotion skills by imitating animals", *arXiv preprint arXiv: 2004.00784*, 2020.

[13] T. Azayev and K. Zimmerman, "Blind hexapod locomotion in complex terrain with gait adaptation using deep reinforcement learning and classification", *Journal of Intelligent & Robotic Systems*, pp. 1–13, 2020.

[14] V. Tsounis, M. Alge, J. Lee, F. Farshidian, and M. Hutter, "Deepgait: Planning and control of quadrupedal gaits using deep reinforcement learning", *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3699–3706, 2020.

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *arXiv preprint arXiv:1707.06347*, 2017.

[16] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, *et al.*, "Solving rubik's cube with a robot hand", *arXiv preprint arXiv:1910.07113*, 2019.

[17] K. De Jong, "Evolutionary computation: a unified approach", in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 185–199.

[18] J. K. Pugh, L. B. Soros, and K. O. Stanley, "Quality diversity: A new frontier for evolutionary computation", *Frontiers in Robotics and AI*, vol. 3, p. 40, 2016.

[19] J.-B. Mouret and J. Clune, "Illuminating search spaces by mapping elites", *arXiv preprint arXiv:1504.04909*, 2015.

[20] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*, 3. MIT press Cambridge, MA, 2006, vol. 2.

[21] Q. Du, V. Faber, and M. Gunzburger, "Centroidal voronoi tessellations: Applications and algorithms", *SIAM review*, vol. 41, no. 4, pp. 637–676, 1999.

[22] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search", in *International conference on computers and games*, Springer, 2006, pp. 72–83.

[23] D. Tarapore, J. Clune, A. Cully, and J.-B. Mouret, "How do different encodings influence the performance of the map-elites algorithm?", in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 173–180.

[24] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning", *arXiv preprint arXiv: 1712.06567*, 2017.

[25] C. Colas, V. Madhavan, J. Huizinga, and J. Clune, "Scaling map-elites to deep neuroevolution", in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 2020, pp. 67–75.

[26] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning", *arXiv preprint arXiv: 1703.03864*, 2017.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning", *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] M. Popova, O. Isayev, and A. Tropsha, "Deep reinforcement learning for de novo drug design", *Science advances*, vol. 4, no. 7, eaap7885, 2018.

[29] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning", in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016, pp. 50–56.

[30] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994.

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

[32] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm", *arXiv preprint arXiv:1712.01815*, 2017.

[33] J. Achiam, "Spinning Up in Deep Reinforcement Learning", 2018.

[34] C. J. Watkins and P. Dayan, "Q-learning", *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[35] S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game starcraft: Broodwar", in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2012, pp. 402–408.

[36] A. R. Tavares and L. Chaimowicz, "Tabular reinforcement learning in real-time strategy games via options", in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2018, pp. 1–8.

[37] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation", *arXiv preprint arXiv:1506.02438*, 2015.

[38] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization", in *International conference on machine learning*, 2015, pp. 1889–1897.

[39] X. B. Peng, G. Berseth, K. Yin, and M. Van De Panne, "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning", *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–13, 2017.

[40] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne, "Deepmimic: Example-guided deep reinforcement learning of physics-based character skills", *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–14, 2018.

[41] F. Abdolhosseini, H. Y. Ling, Z. Xie, X. B. Peng, and M. van de Panne, "On learning symmetric locomotion", in *Motion, Interaction and Games*, 2019, pp. 1–10.

[42] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor", *arXiv preprint arXiv:1801.01290*, 2018.

[43] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", in *International conference on machine learning*, 2016, pp. 1928–1937.

[44] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[46] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners", *arXiv preprint arXiv:2005.14165*, 2020.

[47] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[48] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 23–30.

[49] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization", in *2018 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2018, pp. 1–8.

[50] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, *et al.*, "Learning dexterous in-hand manipulation", *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[51] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition", *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[52] M. C. Mozer, "Neural net architectures for temporal sequence processing", in *Santa Fe Institute Studies in the Sciences of Complexity-Proceedings Volume-*, ADDISON-WESLEY PUBLISHING CO, vol. 15, 1993, pp. 243–243.

[53] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[54] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "RL^2: Fast reinforcement learning via slow reinforcement learning", *arXiv preprint arXiv:1611.02779*, 2016.

[55] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis, "Reinforcement learning, fast and slow", *Trends in cognitive sciences*, vol. 23, no. 5, pp. 408–422, 2019.

[56] D. Bossens, J.-B. Mouret, and D. Tarapore, "Learning behaviour-performance maps with meta-evolution", in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2020.

[57] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study", *arXiv preprint arXiv:2006.05990*, 2020.

[58] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[59] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[60] A. Stooke and P. Abbeel, "Rlpyt: A research code base for deep reinforcement learning in pytorch", *arXiv preprint arXiv:1909.01500*, 2019.

[61] H. Liu and H. Iba, "A hierarchical approach for adaptive humanoid robot control", in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, IEEE, vol. 2, 2004, pp. 1546–1553.

[62] J. Clune, B. E. Beckmann, C. Ofria, and R. T. Pennock, "Evolving coordinated quadruped gaits with the hyperneat generative encoding", in *2009 iEEE congress on evolutionary computation*, IEEE, 2009, pp. 2764–2771.

[63] D. Goepp, K. Chatzilygeroudis, and E. Dalin, *ROS integration for hexapods*, Oct. 2018.

[64] *Servo e-manual*, MX-28T/R/AT/AR, Rev. 2, Robotis.

[65] E. Coumans and Y. Bai, *Pybullet, a python module for physics simulation for games, robotics and machine learning*, `http://pybullet.org`, 2019.

[66] S. Madgwick, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays", *Report x-io and University of Bristol (UK)*, vol. 25, pp. 113–118, 2010.

[67] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. Leonard, "Past, present, and future of simultaneous localization and mapping: Towards the robust-perception age", *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[68] J. J. Craig, *Introduction to robotics: mechanics and control, 3/E*. Pearson Education India, 2009.

[69] D. Singh and B. Singh, "Investigating the impact of data normalization on classification performance", *Applied Soft Computing*, p. 105 524, 2019.

[70] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning", *arXiv preprint arXiv:1509.02971*, 2015.

[71] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates", in *2017 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2017, pp. 3389–3396.

[72] I. Mordatch, K. Lowrey, G. Andrew, Z. Popovic, and E. V. Todorov, "Interactive control of diverse complex characters with neural networks", in *Advances in Neural Information Processing Systems*, 2015, pp. 3132–3140.

[73] X. B. Peng and M. van de Panne, "Learning locomotion skills using deeprl: Does the choice of action space matter?", in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2017, pp. 1–13.

[74] S. Butterworth *et al.*, "On the theory of filter amplifiers", *Wireless Engineer*, vol. 7, no. 6, pp. 536–541, 1930.

[75] K. Perlin, "An image synthesizer", *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[76] K. Perlin, “Improving noise”, in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 681–682.

[77] J. A. Hartigan, *Clustering algorithms*. John Wiley & Sons, Inc., 1975.

[78] J. Platt, “A resource-allocating network for function interpolation”, *Neural computation*, vol. 3, no. 2, pp. 213–225, 1991.

Appendix A

Implementation Details

A.1 Perlin Noise

Instead of calculating the value at each coordinate randomly, gradient noise methods only compute random values on a lattice, and use interpolation to calculate the values between the lattice points. In the case of Perlin Noise [75, 76] an *integer lattice* is used, which encapsulates all points in the two-dimensional space with integer coordinates $(i \quad j)^T$ for $i, j \in \mathbb{Z}$ (see Figure A.1). Each point on the integer lattice is associated with a gradient vector $g_{i,j}$ which is uniformly sampled from the set $\{(-1 \quad -1)^T, (-1 \quad 1)^T, (1 \quad 1)^T, (1 \quad -1)^T\}$ (see Figure A.2).

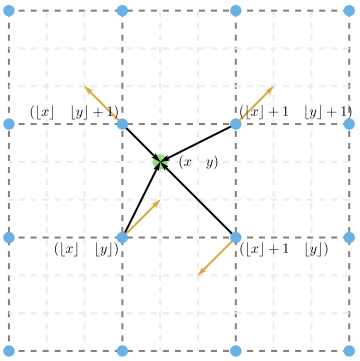


Figure A.1: The integer lattice grid used in Perlin noise.

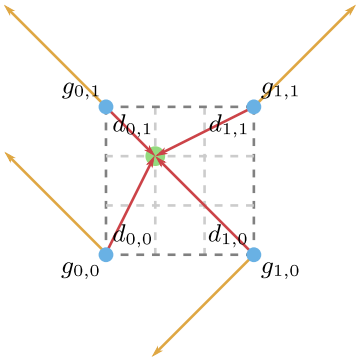


Figure A.2: A zoomed view of a grid cell from the lattice grid.

For each point $(x \quad y)^T$ that is not on the integer lattice, four vectors $d_{i,j}$, for $i, j \in \{0, 1\}$, are calculated according to Equations (A.1 - A.4) that capture the difference between the coordinates of the point itself, and the coordinates of the four nearest points on the integer lattice (see Figure A.2).

$$d_{0,0}(x, y) = (x \quad y)^T - (\lfloor x \rfloor \quad \lfloor y \rfloor)^T \tag{A.1}$$
$$d_{0,1}(x, y) = (x \quad y)^T - (\lfloor x \rfloor \quad \lfloor y \rfloor + 1)^T \tag{A.2}$$
$$d_{1,0}(x, y) = (x \quad y)^T - (\lfloor x \rfloor + 1 \quad \lfloor y \rfloor)^T \tag{A.3}$$
$$d_{1,1}(x, y) = (x \quad y)^T - (\lfloor x \rfloor + 1 \quad \lfloor y \rfloor + 1)^T \tag{A.4}$$

Then, four vertical displacement values $\delta_{i,j}$, for $i, j \in \{0, 1\}$, are computed as the

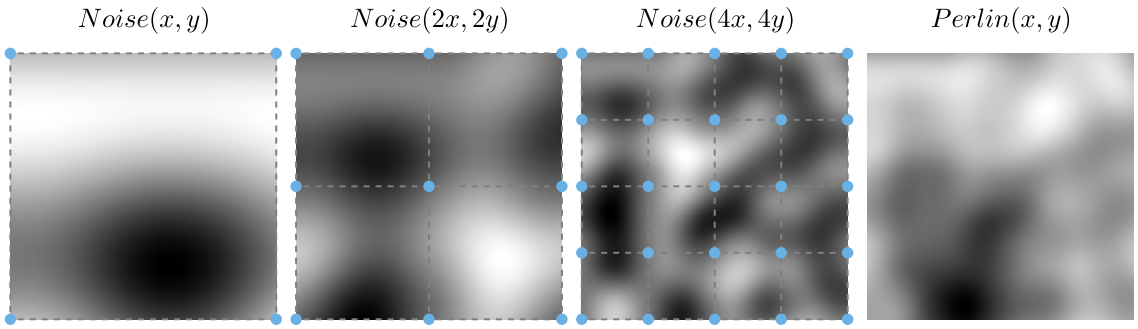


Figure A.3: Example patterns generated with Perlin noise.

dot product between the four nearest gradient vectors $g_{i,j}$ and the four difference vectors $d_{i,j}$ from above (see Equations (A.5 - A.8)).

$$\delta_{0,0}(x, y) = d_{0,0}(x, y) \cdot g_{\lfloor x \rfloor, \lfloor y \rfloor} \tag{A.5}$$
$$\delta_{0,1}(x, y) = d_{0,1}(x, y) \cdot g_{\lfloor x \rfloor, \lfloor y \rfloor + 1} \tag{A.6}$$
$$\delta_{1,0}(x, y) = d_{1,0}(x, y) \cdot g_{\lfloor x \rfloor + 1, \lfloor y \rfloor} \tag{A.7}$$
$$\delta_{1,1}(x, y) = d_{1,1}(x, y) \cdot g_{\lfloor x \rfloor + 1, \lfloor y \rfloor + 1} \tag{A.8}$$

Lastly, the weighted sum of the vertical displacement values determines the noise value at the point $(x \quad y)$. The weights are computed according to the fade function in Equation (A.10).

$$\psi(x) = 6x^5 - 15x^4 + 10t^3 \tag{A.9}$$
$$\Psi(x, y) = \psi(x)\psi(y) \tag{A.10}$$

The fade function is an s-shaped curve which gives a weight of 1 when the lattice points are close to $(x \quad y)^T$, and smoothly reduce the weight as $(x \quad y)^T$ get further from the lattice points. This function provides the smooth interpolation between the different gradients on the integer lattice. The complete noise function is:

$$\begin{aligned} Noise(x, y) = {} & \Psi(1 - \hat{x}, 1 - \hat{y})\delta_{0,0}(x, y) \\ & + \Psi(\hat{x}, 1 - \hat{y})\delta_{1,0}(x, y) \\ & + \Psi(1 - \hat{x}, \hat{y})\delta_{0,1}(x, y) \\ & + \Psi(\hat{x}, \hat{y})\delta_{1,1}(x, y) \end{aligned} \tag{A.11}$$

Where $\hat{x} = x - \lfloor x \rfloor$ and $\hat{y} = y - \lfloor y \rfloor$ are the coordinates of the point within the bounds of the four nearest integer lattice points. Textures generated by the $Noise(\cdot, \cdot)$ function are shown in the first three columns of Figure A.3. Perlin noise is generated by summing k octaves of the $Noise(\cdot, \cdot)$ function as in Equation (A.12).

$$Perlin(x, y) = \sum_{i=0}^{k} 0.5^i Noise(2^i x, 2^i y) \tag{A.12}$$

The last texture in Figure A.3 is generated using the $Perlin(\cdot, \cdot)$ function with $k = 3$.

A.2 Hyperparameters

Table A.1 shows the hyperparmeter used for PPO training for all agents. Table A.2 shows the hyperparmeter used for control and definition of the ADR distributions.

Hyperparameter	*FC*, *FC + ADR*	*LSTM*, *LSTM + ADR*	*CNN*, *CNN + ADR*
discount factor γ	0.99	0.99	0.99
GAE λ	0.95	0.95	0.95
entropy regularisation coeff.	5e-3	5e-3	5e-3
PPO clipping parameter ϵ	0.4	0.4	0.4
optimiser	Adam	Adam	Adam
learning rate	3e-5	3e-5	3e-5
batch size (chunks $\times$ transitions)	40 $\times$ 256	40 $\times$ 256	40 $\times$ 256
minibatch size	256	512	256
sample reuse	20	15	20
value loss weight	0.5	0.5	0.001
L2 regularisation weight	1e-5	1e-5	1e-5
gradient norm clipping	0.5	0.5	0.5

Table A.1: Hyperparameters used for PPO on a machine with 40 CPU cores and 1 NVIDIA GeForce GTX 1080 GPU.

Parameter	Low	High	Δ_i	m_i	t_H^i	t_L^i	Limits	Symmetric
Stone Density	0.0	0.0	0.010	10	0.2	0.7	$[0, +\infty)$	✗
Slide Friction	1.0	1.0	0.002	10	0.2	0.7	$[0.5, 1.5]$	✗
Amplitude	0.0	0.0	0.100	10	0.2	0.7	$[0, +\infty)$	✗
Elevation Gain	0.0	0.0	0.050	10	0.2	0.7	$[-1, +\infty)$	✗
Stair Height	0.0	0.0	0.010	10	0.2	0.7	$[-0.2, +\infty)$	✗
Robot y	0.0	0.0	0.100	10	0.2	0.7	$[-1.5, 1.5]$	✓
Robot Yaw	0.0	0.0	0.100	10	0.2	0.7	$[-3.14, 3.14]$	✓
Gravity	-9.8	-9.8	0.200	10	0.2	0.7	$[-15, -5]$	✗
Random Obs. Bias	0.0	0.0	0.005	10	0.2	0.7	$(-\infty, \infty)$	✓
Uncorr. Obs. Noise	0.0	0.0	0.005	10	0.2	0.7	$(-\infty, \infty)$	✓
Power Reduction	0.0	0.0	0.005	10	0.2	0.7	$[0, 1]$	✗

Table A.2: Hyperparameters used for ADR.

A.3 Empirical Maximum of ADR Boundaries

Parameter	Boundaries
Stone Density	[0.000, 0.620]
Slide Friction	[0.820, 1.150]
Amplitude	[0.000, 0.800]
Elevation Gain	[-0.950, 1.600]
Stair Height	[-0.140, 0.070]
Robot y	[-1.500, 1.500]
Robot Yaw	[-3.141, 3.141]
Gravity	[-15.000, -6.400]
Random Obs. Bias	[-0.250, 0.250]
Uncorr. Obs. Noise	[-0.430, 0.430]
Power Reduction	[0.000, 0.430]

Table A.3: The widest boundaries for each parameter that have been ever achieved during training or evaluation by any agent.

A.4 Description of CVT and Damage Codes

Damage Code	Description
D_1	Leg 1 is missing and its sensors read zeros
D_2	Leg 2 is missing and its sensors read zeros
D_3	Leg 3 is missing and its sensors read zeros
D_4	Leg 4 is missing and its sensors read zeros
D_5	Leg 5 is missing and its sensors read zeros
D_6	Leg 6 is missing and its sensors read zeros
D_7	Servos s_1^1, s_2^1, s_3^1 have no power but all sensors work correctly
D_8	Servos s_1^2, s_2^2, s_3^2 have no power but all sensors work correctly
D_9	Servos s_1^3, s_2^3, s_3^3 have no power but all sensors work correctly
D_{10}	Servos s_1^4, s_2^4, s_3^4 have no power but all sensors work correctly
D_{11}	Servos s_1^5, s_2^5, s_3^5 have no power but all sensors work correctly
D_{12}	Servos s_1^6, s_2^6, s_3^6 have no power but all sensors work correctly
D_{13}	Legs 1 and 5 are missing and their sensors read zeros
D_{14}	Legs 1 and 6 are missing and their sensors read zeros
D_{15}	Legs 2 and 5 are missing and their sensors read zeros
D_{16}	Legs 2 and 6 are missing and their sensors read zeros
D_{17}	Legs 3 and 5 are missing and their sensors read zeros
D_{18}	Legs 4 and 6 are missing and their sensors read zeros

Table A.4: Description of the damage codes.

Centroid Code	Stone Density	Slide Friction	Amplitude	Elevation Gain	Robot y	Robot Yaw	Gravity	Const. Obs. Noise Bias	Uncorr. Obs. Noise	Power Reduction
C_B^1	0.43214	1.04341	0.5416	0.82990	-0.5376	1.1904	-12.4028	0.0970	0.07396	0.27262
C_B^2	0.18910	1.01272	0.5064	-0.17990	0.5264	-1.1966	-12.3942	0.0985	0.15308	0.29111
C_B^3	0.43028	0.95728	0.2944	0.82735	-0.5376	1.1966	-9.0230	-0.0980	-0.15480	0.13846
C_B^4	0.43090	0.92659	0.2584	0.82480	0.5404	-1.1842	-12.3770	0.0990	-0.11524	0.17544
C_B^5	0.41478	1.05067	0.2408	-0.10595	0.5376	1.1966	-11.9126	-0.0475	0.16770	0.13029
C_B^6	0.18848	0.92065	0.2456	0.82480	-0.0252	0.7936	-12.3770	-0.0965	0.16426	0.29713
C_B^7	0.22258	1.04935	0.2440	0.55705	0.5432	1.2028	-9.2294	0.0865	-0.16856	0.29971
C_B^8	0.18848	1.04869	0.5552	0.81205	0.3640	-0.0930	-12.3770	-0.0970	-0.16340	0.13287
C_B^9	0.41540	0.92032	0.5576	-0.10850	0.5432	1.2090	-11.4998	-0.0690	-0.16942	0.30057
C_B^{10}	0.20584	1.04968	0.2416	0.75850	-0.5572	-1.1842	-9.8916	0.0710	0.16770	0.13029
C_B^{11}	0.43338	1.02922	0.4672	0.82735	0.5404	-1.1780	-9.0144	-0.0990	0.14964	0.29154
C_B^{12}	0.43090	0.92164	0.2448	-0.16970	-0.3668	0.0930	-9.0316	0.0975	0.16598	0.29670
C_B^{13}	0.36828	1.04869	0.2424	-0.02690	-0.5404	-1.2028	-12.1792	-0.0845	-0.17028	0.29971
C_B^{14}	0.43028	1.04902	0.5568	-0.16715	0.0392	-0.8060	-9.0316	0.0955	-0.16684	0.13330
C_B^{15}	0.39804	0.92032	0.5568	0.09040	-0.5376	-1.1904	-12.1620	-0.0865	0.16942	0.13072
C_B^{16}	0.18786	0.92692	0.2584	-0.18500	0.5376	-1.1718	-8.9886	-0.0990	-0.07654	0.15824
C_B^{17}	0.25172	0.92065	0.5584	0.67690	0.5376	1.2090	-9.2208	0.0850	0.16856	0.13115
C_B^{18}	0.18848	1.04374	0.5400	-0.17480	-0.5348	1.1842	-9.0230	-0.0990	0.11610	0.25499
C_B^{19}	0.20274	0.91966	0.5560	0.76105	-0.5376	-1.2028	-9.4874	0.0470	-0.16770	0.29842
C_B^{20}	0.18600	0.94078	0.3320	-0.17480	-0.5404	1.1904	-12.3942	0.0975	-0.15136	0.13932

Table A.5: Description of the cluster centre codes on terrains with bumps.

Centroid Code	Stone Density	Slide Friction	Stair Height	Robot y	Robot Yaw	Gravity	Const. Obs. Noise Bias	Uncorr. Obs. Noise	Power Reduction
C_S^1	0.43772	1.05397	0.00847	0.5908	-1.3144	-9.4874	0.0845	0.11524	0.28853
C_S^2	0.36518	1.04869	0.00553	-0.5348	1.2028	-9.0402	-0.0980	0.16684	0.13115
C_S^3	0.18228	0.91702	0.00889	0.5824	1.3082	-9.2380	0.0710	0.14706	0.27090
C_S^4	0.18414	0.91636	-0.07910	-0.5936	1.3020	-11.8782	-0.0870	-0.11524	0.14190
C_S^5	0.18228	1.05496	-0.07847	-0.5936	-1.2958	-9.2380	0.0655	0.14534	0.27563
C_S^6	0.18290	1.04308	-0.06356	0.3472	1.0912	-8.8854	0.1015	-0.17974	0.12384
C_S^7	0.43710	0.92692	-0.00665	-0.3556	-1.0850	-12.5318	-0.1040	0.17802	0.30573
C_S^8	0.30132	0.92032	-0.07532	0.5544	-1.2090	-9.0144	-0.0990	0.16770	0.13029
C_S^9	0.18228	1.04308	-0.06167	0.3808	1.0788	-12.4888	-0.1055	0.18146	0.30401
C_S^{10}	0.18476	0.93913	0.00133	-0.4872	-0.7936	-8.8682	-0.1055	-0.17802	0.30444
C_S^{11}	0.43772	0.91438	0.00868	0.5936	1.2834	-12.1448	-0.0660	-0.14362	0.15437
C_S^{12}	0.18290	1.05529	0.01015	0.5768	-1.2834	-11.8266	-0.0845	-0.12556	0.14190
C_S^{13}	0.43710	1.02724	-0.07196	0.4956	0.8370	-8.9198	-0.1030	-0.17974	0.30616
C_S^{14}	0.43710	0.92725	-0.00812	-0.3836	-1.0788	-8.9112	0.1070	-0.18060	0.12599
C_S^{15}	0.43648	0.91570	-0.07973	-0.5768	1.2896	-9.5476	0.0850	0.12384	0.28896
C_S^{16}	0.43648	1.02955	-0.07175	0.4900	0.7874	-12.5146	0.1055	0.17888	0.12556
C_S^{17}	0.25482	0.92098	-0.07511	0.5404	-1.1842	-12.3770	0.0980	-0.16770	0.29842
C_S^{18}	0.31620	1.04968	0.00616	-0.5432	1.2214	-12.3684	0.0975	-0.16942	0.30014
C_S^{19}	0.18166	0.94276	0.00133	-0.4900	-0.8556	-12.4888	0.1025	0.18146	0.12298
C_S^{20}	0.43710	1.05331	-0.07952	-0.5852	-1.3206	-12.1706	-0.0695	-0.14534	0.15781

Table A.6: Description of the cluster centre codes on terrains with stairs.