Learning Versatile Motor Skills For Dynamic Quadruped Locomotion

Zhaocheng Liu

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

Robot locomotion is becoming more powerful and popular these days. Especially, locomotion for the legged robot has brought the mobility of the autonomous agents to another level. This project presents a deep reinforcement learning approach that is capable of learning a partially observable locomotion policy with the Proximal Policy Optimisation algorithm. We propose a novel reward shaping design known as the Dynamic Reward Strategy (DRS), which introduces great improvements of locomotion in terms of velocity stability, robustness in different terrain and power efficiency. The proposed method learns high-performance locomotion using a lightweight Deep Neural Network without the dependency of any form of the history data. The performance of this approach is evaluated in the OpenAI gym simulation environment and has scored an over 90% success rate in all challenging terrain.

Acknowledgements

I would like to express my sincere gratitude towards my supervisor, Dr Zhibin Li, for his guidance and support throughout this project, as well as, Mr Fernando Acero for his valuable discussions and support.

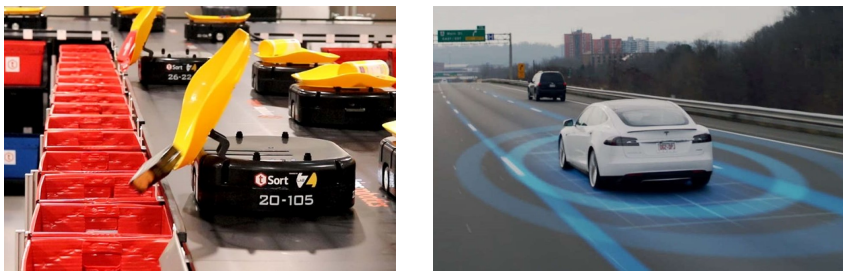
Table of Contents

Chapter 1

Introduction

1.1 Background

Autonomy has been a hot topic in the recent decades. It has brought huge benefit to humanity. For example, in an autonomous sorting warehouse[1], the "mini yellow" robot system has reduced 60% of the labour cost while only requiring about 1/4 space if a cross-belt sorting system were to be installed. See figure 1.1a, "mini yellow" robots are sorting parcels into baskets.



(a) "Mini yellow" robots in operation (b) Autonomous car by Tesla

Figure 1.1: Applications of the autonomous robots (images from the internet)

Another great example is the autonomous cars by Tesla (figure 1.1b), which is considered as a huge success in the autonomous transportation industry. Tesla cars, powered by the Artificial Intelligence, use the Deep Neural Networks to analyse roads, objects and make right decisions for driving the vehicle. The innovation of Tesla has brought many convenience into to people's life. The safety is also well guaranteed by Tesla. According to Forbes[2], the accident rate is "once per 4.59 million miles", which is lower than the US average, "once per 479,000 miles" (3rd quarter, 2020).

[1]Source:https://asia.nikkei.com/Business/Startups/China-robotics-startup-offers-automation-in-distribution-centers

[2]Source:https://www.forbes.com/sites/bradtempleton/2020/10/28/new-tesla-autopilot-statistics-show-its-almost-as-safe-driving-with-it-as-without/?sh=1b08884d1794

Now talking about the legged robot locomotion, which has taken the mobility and maneuver capability of the autonomous agents up to another level [1]. This has brought the autonomy into a wider range of applications. For example, the BigDog by Boston Dynamics ([2], in 2008), which is a novel approach of a quadrupedal robot that imitates the animal gait and is capable of carrying heavy weights while climbing on a rocky slopes up to 60 degrees, or, walking on the flat ground at about of 3 km/h under the external force disturbances.

In 2020, *T. Klamt* et al. [3] proposed a novel robot, Centauro robot, a quadruped robot on wheels with two robot arms, which is able to operate in inaccessible and dangerous environments. This robot combines the mobility of the legged robot and the manipulation ability of the robot arms, and has enabled a possibility of many challenging applications, including the autonomous disaster rescue.

In a summary, robot locomotion has a great potential in many autonomous applications for benefiting the humanity. In this project, we only focus on the locomotion of the quadrupedal robot.

1.2 Related works

1.2.1 Multi-expert learning of adaptive legged locomotion

There has been many successful approaches of the quadrupedal robot locomotion that are skillful and robust. In 2020, *C. Yang, K. Yuan* et al. proposed a Multi-Expert Learning Architecture (MELA) that learns to generate adaptive skills from a group of representative expert skills [4]. MELA uses a Gating Neural Network to dynamically synthesise the output from each expert Deep Neural Network (DNN) dynamically, according to the situation. This novel approach has shown a robust multi-skilled locomotion on a real robot, including trotting, steering and fall recovery.

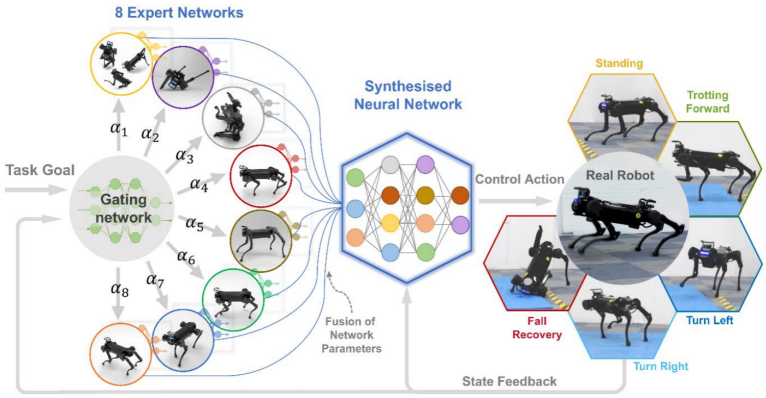


Figure 1.2: The architecture of MELA

In a summary, MELA has proven the success of learning a robust locomotion at the joint-level control, also, learning locomotion skills with individual DNNs. Their approach adopts 25 Hz sampling rate and 1000 Hz impedance control. The 25 Hz sampling rate is also adopted in my project, discussed latter.

1.2.2 Learning quadrupedal locomotion over challenging terrain

Also in 2020, *J. Lee* et al. proposed a novel approach of robust and adaptive blind quadrupedal locomotion model [5]. The proposed controller is able to reliably drive the ANYmal quadrupedal robot travelling through a variety of difficult terrain, including mud, sand, rubble, thick vegetation, snow, running water, and a variety of other off-road terrain. This approach uses a temporal convolutional network (TCN) that generates trajectories based on an extended history of proprioceptive states, which consists of the measurements from joint encoders and an inertial measurement unit (IMU). See in the figure 1.3, a snapshot from a experiment video[3].

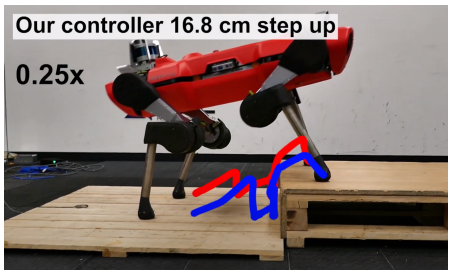


Figure 1.3: A snapshot of the robot walks on a flat surface, and climbs up a stair. The front leg trajectories are traced with red and blue lines. When the front legs hits and have been stopped, the algorithm infers there is a step ahead. Then, the agent comes up with a solution and has eventually climbed up the stair. (Image from the video)

More detail of their approach, they have introduced a teacher policy, which is trained with the ground-truth information of the terrain and the robot's contact information, hence, the teacher policy quickly becomes an expert. Then, it is used to guide the learning of a student controller that only observe through the on-board sensors. After learning, the student policy is directly deployed on real robots and is proven to be a success.

For further analysis, the teacher policy in their approach is playing an important role of extracting useful features from the fully-observable environment. Later, when teaching the student policy, the features are further extracted and learned with limited observation. This ideology serves as a hint to my project as: a set of well-abstracted/designed features is one of the key factors of a successful and robust locomotion learning.

1.3 Motivation

As many people have done in the past, a robust locomotion model usually relies on a complex model with some form of the history data as the memory. In this project, the aim is to develop a methodology to train an online locomotion model with the aid of the vision input, but, without the dependence of memory. Further more, this approach should be concise, fast to learn and robust in a rugged terrain with obstacles.

[3] Video:`https://www.youtube.com/watch?v=tPixnjLbTvE`

Chapter 2

Background

2.1 Classical robot control methods

This chapter discusses the theories and background knowledge that are related to this project, such as, (I) what is a "robot", (II) the classical robot control methods, with an intuitive mass-spring-damper example, (III) the Forward Dynamics algorithm which is crucial for the execution of the experiments, (IV) a brief introduction to the Deep Reinforcement Learning and the Proximal Policy Optimisation algorithm. And lastly, the definition of the problem of this project.

2.1.1 Preliminaries

A *robot* is a mechanical structure consisting of many "bodies", called ***links***, connected by ***joints*** [6]. There are many types of joints, as shown in the figure 2.1. In this project, every link is considered as a "rigid body", which means they have a fixed shape and mass, and undeformable. Also, every joint is assumed a "revolute joint". For each joint, there is an electric motor embedded, which generates torque to drive the attached link.

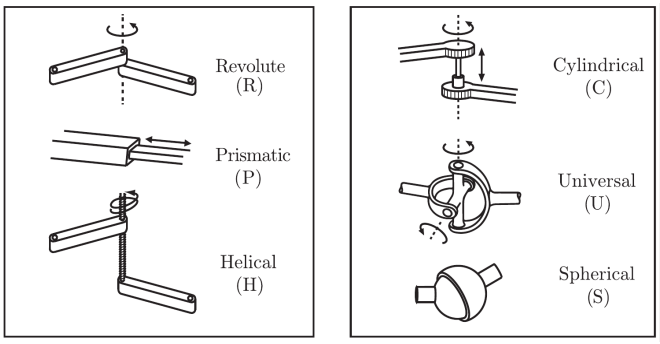


Figure 2.1: Example of different types of joints

The ***degree of freedom (DoF)*** of a robot is defined as the minimum number n of real-valued coordinates needed to represent the configuration of the robot. For an open-

loop robot like the quadrupedal robot in this project, its DoF is the total number of joints. The ***configuration space (C-space)***, $\mathcal{Q}$, is defined as the n-dimensional space containing all possible configurations of the robot, $\mathcal{Q} \subset \mathbb{R}^n$.

The position of a joint is denoted by θ, which is the angle measured from its rest position. The ***configuration*** of a robot is a representation of all the joint positions in the form of a vector. For a robot with n joints, its configuration vector, $\mathbf{q} \in \mathcal{Q}$, is defined as:

$$\mathbf{q} = [\theta_1, \theta_2, \ldots, \theta_n]^T \tag{2.1}$$

Accordingly, the joint velocity vector is $\dot{\mathbf{q}}$ and the joint acceleration vector is $\ddot{\mathbf{q}}$ which are, respectively, the first and second derivatives of the configuration vector:

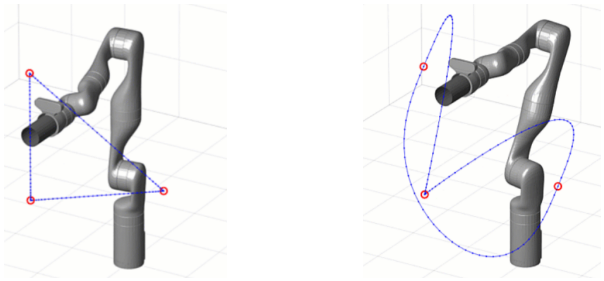
$$\text{Joint velocity } \dot{\mathbf{q}} = [\dot{\theta}_1, \dot{\theta}_2, \ldots, \dot{\theta}_n]^T, \text{ and} \tag{2.2}$$

$$\text{Joint acceleration } \ddot{\mathbf{q}} = [\ddot{\theta}_1, \ddot{\theta}_2, \ldots, \ddot{\theta}_n]^T. \tag{2.3}$$

The ***workspace*** of a robot is the set of points where its end-effector can reach. The "end-effector" is a link or a device at the end of a structure. It can be the finger tip of a robot hand, a rubber gripper or others. The ***task pace*** is the space in which the task is expressed and independent to the workspace of any robot, e.g. for painting, the task space is the 2-D surface of the canvas, but the workspace can be the 3-D space around the canvas.

2.1.2 Trajectory planning

A trajectory is a sequence of joint positions and velocities at each timestep. It is an interpolation from the starting configuration to the desired configuration. Depends on the level of abstraction, this procedure can be done in either the C-space or the task space. An example of a "C-space trajectory" and a "task space trajectory":



(a) (b)

Figure 2.2: A robot arm with its base fixed on the ground, its end-effector is moving according to some trajectories: the task space trajectory (a): interpolating in 3-D. the C-space trajectory (b): interpolating the configuration. The trajectories start and end at the bottom left point. The length of the timestep is a fixed duration, $\delta t = 1/f$, where f is the trajectory sampling frequency at which the controller plan the trajectories. At each timestep, the end-effector is at each point on the blue line which is a trace of the trajectory.

In figure 2.2, the end-effector is moving at a constant speed. For the task space trajectory on the left, there is a sudden change of the end-effector velocity at each corner of the triangle. That means some of the joints are suffering a sudden change of torques, which is harmful to the hardware on a real robot. On the other hand, the C-space trajectory, by design, is always having a smooth joint transition in time. However, the behaviour of the task space trajectory is more intuitive and organized.

2.1.3 Online vs. Offline planning

Planning trajectories entirely before the execution is call *offline* trajectory planning. Or, it can be done successively along with the motion, which is called the *online* trajectory planning, where a fixed length of the future trajectory $l \geq 1$ is generated at each timestep.

Planning in task space is generally much slower, because "reverse-engineering" the configuration given only the end-effector position requires extra calculation. The "reverse-engineering" algorithm is called the *Inverse Kinematics*, which is irrelevant to this project. Therefore, C-space trajectory planning is better suited to an online planning application.

2.1.4 Feedback control

The *Proportional–Integral–Derivative controller* ***(PID controller)*** is proven to be a novel feedback control model and is widely used in various applications [6]. It manipulates the joints to desired positions by minimising the error in each joint, $\mathbf{q}_e$. For a revolute joint, the PID controller is defined as:

$$\mathbf{u}(t) = K_p \mathbf{q}_e + K_i \int \mathbf{q}_e(t)\, dt + K_d \dot{\mathbf{q}}_e \tag{2.4}$$

where $\mathbf{u}(t) \in \mathbb{R}^n$ is a vector of the output joint torques at time t, the control gains K_p, K_i and K_d are positive constants, $\mathbf{q}_{ref}$ is the desired joint configuration, the joint error $\mathbf{q}_e = \mathbf{q}_{ref}(t) - \mathbf{q}(t)$ and the joint velocity error $\dot{\mathbf{q}}_e = \dot{\mathbf{q}}_{ref}(t) - \dot{\mathbf{q}}(t)$.

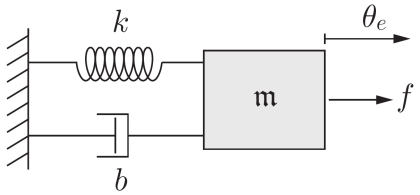


Figure 2.3: A linear mass-sparing-damper model: the mass can only move in 1 DoF, so, we directly use θ to denote its position. There is a small but non-zero external force f applied to the mass in the positive direction. The mass is attached to a spring, k, and a viscous damper, b. Assume the mass is θ_e away from its rest position, where $|\theta_e| > 0$ and the desired velocity $\dot{\theta}_{ref} = 0$ m/s.

For a better illustration of the PID control, here we introduce a simple example of a mass-spring-damper model (2.3). In a PID controller, the K_p term is similar to the

spring that applies a force onto the mass proportional to the error. The K_d term is similar to the viscous damper that always reduces the speed of the mass.

The response curves illustrates the resulting configuration error of the mass, under the control of the PID controller with different settings, shown in the figure 2.4. For a P-controller ($K_p > 0$, and $K_p = K_i = 0$), the mass becomes an oscillator with a fixed amplitude. For a PD-controller ($K_p > 0, K_p > 0$, and $K_i = 0$), the mass eventually comes to rest after a few oscillations, but there remains a small steady-state error due to the external force. For a PID-controller (with correct parameters $K_p > 0, K_p > 0$ and $K_i > 0$), the mass always returns to its rest position.

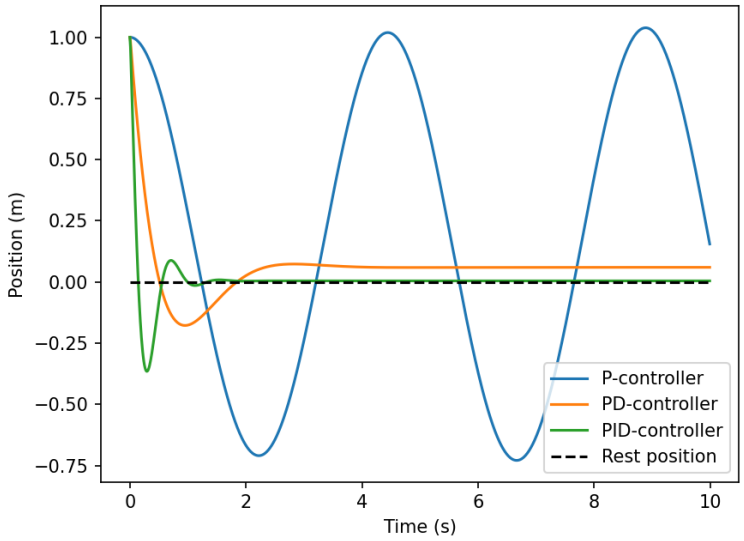


Figure 2.4: Response of a: P-controller (blue line), PD-controller (yellow line), a PID-controller (green line). And the rest position without the external force applied (dashed black line)

2.2 Forward Dynamics

On a real robot, the physics is in responsible of governing the robotic dynamics. However, in a simulation environment, the physics has to be simulated by an algorithm called the ***Forward Dynamics (FD)***. The FD algorithm determines the acceleration $\ddot{\mathbf{q}}$ of the robot, given the state $(\mathbf{q}, \dot{\mathbf{q}})$ and the joint forces and torques.

The rigid bodies can be abstracted as point-masses in the FD calculation. The physics property *moment of inertia (inertia)*, I, of the rigid body is defined as:

$$I = mr^2 \tag{2.5}$$

Where, m is the mass of the rigid body and r is the displacement from its center of mass (CoM) to the joint from which the rigid body receives the torque.

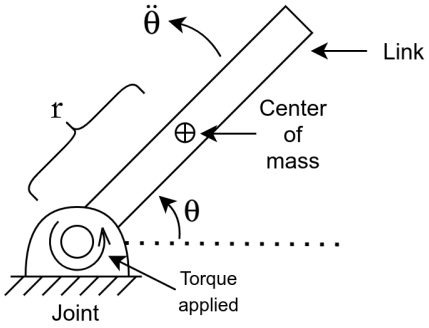


Figure 2.5: Example of a rigid body (the link) attached to a joint that is fixed on the ground. The link is at θ degrees from the ground level, with its CoM marked in the diagram. The A torque is applied at the joint.

As illustrated in 2.5, the actuator applies a torque at the joint, the link will start to accelerate with an angular acceleration $\ddot{\theta}$.

$$\ddot{\theta} = \frac{\tau}{I} \tag{2.6}$$

The joint velocity is calculated as:

$$\dot{\theta} = \int \ddot{\theta}\, dt \tag{2.7}$$

Given a time period Δt that is small enough, the new angular velocity is approximated:

$$\dot{\theta}_{t+\Delta t} \approx \dot{\theta}_t + \ddot{\theta} \cdot \Delta t \tag{2.8}$$

2.3 Deep reinforcement learning

Deep reinforcement learning (DRL) is a branch of *machine leaning* that learns to make decisions by trial and error , which has shown a great success in many applications till today. For example, "Alpha GO" by DeepMind which beats the world champions of Go chess [7].

2.3.1 The Markov Decision Process

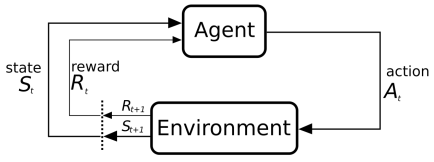


Figure 2.6: DRL learning pattern

DRL is based on the Markov Decision Process (MDP). As shown in the figure 2.6[1], an agent at every timestep is in a state S_t, takes action A_t, receives an immediate reward R_t and transitions to the next state S_{t+1} according to the environment dynamics.

[1]Source: `https://en.wikipedia.org/wiki/Deep_reinforcement_learning`

Mathematically, the MDP is defined as a 7-tuple $\{S, A, O, T, \Omega, R, \gamma\}$, where S, A and O are sets of states, actions, and observations. The transition function $T(s'|s,a)$ defines the distribution over the transition (s,a,s') according to the environment dynamics. The observation function $\Omega(o|s',a)$ defines the distribution over observations $o \in O$ that the agent may receive in the new state s' after action a, and receive an immediate reward. The reward $r_t \sim R(s_t, a_t)$ defines the immediate reward of each state-action pair. The discount factor is defined as $\gamma \in [0,1)$. Finally, The discounted sum of rewards over the planning horizon h (which can be infinite) is $\sum_{t=0}^{h} \gamma^t r_t$.

The objective of DRL is to predict the optimal trajectories in MDPs. For example, Q-learning, an off-policy algorithm, is proven to be successful in many DRL applications, such as playing Atari games at expert level [8]. It directly learns the reward of each state-action pair. This works well in the simple and/or discrete environments, e.g. Atari games, but performs poorly in the high-dimensional continuous environments, where the dynamics becomes much more complex (*Sutton & Barto*, Chapter 13 [9] and [10]).

On-policy algorithms are subject to this challenge and they are proven to be successful [11]. Rather than learning rewards, the on-policy DRL algorithms learn the probabilities of taking each actions in different situations. The ***policy***, π_θ, is defined as a distribution over the actions given a state:

$$\pi_\theta(a|s) = \Pr(a_t = a | s_t = s, \theta) \tag{2.9}$$

This policy function is approximated by a ***multilayer perceptron (MLP)***, a *feedforward Deep Neural Network*, which is proven to be a good function approximator. The inputs are fed into the *input layer* (figure 2.7), propagate through the *hidden layers* and output from the *output layer*. The layers are densely connected between each other, and the flow of data is shown as the arrows in the graph. In each neuron, there is a non-linear activation function.

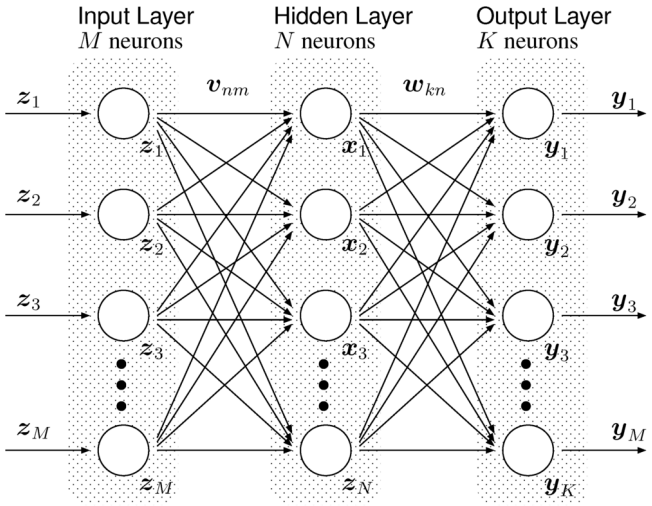


Figure 2.7: An illustration of a MLP with one hidden layer (image from the internet)

When sampling from the policy at time t, the state vector s_t is fed into the MLP and an action a_t is returned from the MLP;

$$a_t = \max_a \pi_\theta(a|s_t). \tag{2.10}$$

By repeating this procedure at each timestep, we are able to generate a MDP trajectory with a variable length in the form:

$$(s_0, a_0, s_1, a_1, s_2, \ \dots\ , s_t, a_t, s_{t+1}, a_{t+1}, s_{t+2}, \ \dots\ ) \tag{2.11}$$

Then, a sequence of the actions from a MDP trajectory can be used for controlling the robot. For example, if each action contains a robot configuration vector, $\mathbf{q}$, then the actions forms a joint trajectory

$$(\mathbf{q}_0, \mathbf{q}_1, \ \dots\ , \mathbf{q}_t, \mathbf{q}_{t+1}, \ \dots\ ) \tag{2.12}$$

2.3.2 Proximal Policy Optimisation algorithm

In 2017, OpenAi proposed a novel off-policy DRL algorithm called ***Proximal Policy Optimization (PPO)*** [11]. This algorithm optimises a "clipped surrogate" objective function using *stochastic gradient ascent (SGA)* [12]. The estimation of the ***policy gradient*** in the SGA is

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t], \tag{2.13}$$

where, π_θ is the stochastic policy and $\hat{A}_t$ is an estimator of the advantage function, which estimates how good an action compared to the average action for a specific state. If the policy is improving, $\hat{A}_t$ is positive, otherwise, $\hat{A}_t$ is negative. The advantage function is defined as:

$$\hat{A} = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.14}$$

$$\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(S_t) \tag{2.15}$$

and T is the length of the trajectory segments, γ is the discount factor, λ is a hyper-parameter of PPO and $V(s)$ is a learned state-value function, e.g. the generalized advantage estimation [13]. The objective function of PPO algorithm is defined as

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{2.16}$$

Where c_1, c_2 are coefficients, L^{CLIP} is the clipped advantage function, whose characteristic is shown in the figure 2.8, L^{VF} is a squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$ and $S[\pi_\theta](s_t)$ is an entropy bonus that ensures sufficient exploration during learning [11]. For more details,

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{2.17}$$

$$r = \frac{\pi_\theta(a_t|st)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.18}$$

Where ε is the clipping parameter usually one of the value in $\{0.1,\ 0.2,\ 0.3\}$ and r is the probability ratio between the old and the new policy. This means that, if the policy deviates too far away from the old one, the probability ratio $r \to 0$. Hence, that policy will be ignored. Regardless the policy is becoming better or worse, r is clipped within $[1-\varepsilon, 1+\varepsilon]$. This mechanism ensures the new policy never deviate too much from the old one.

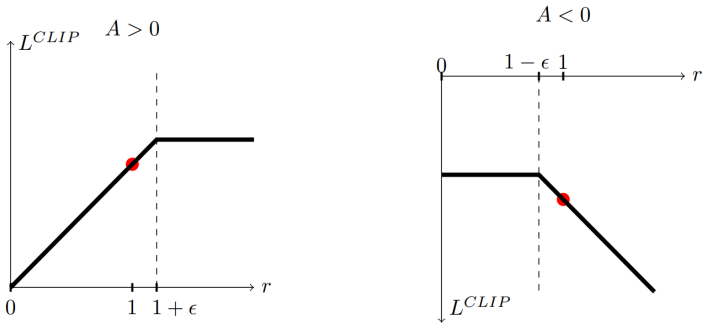


Figure 2.8: Shows the characteristic of L_t^{CLIP} at a single timestep, for positive advantages (left) and negative advantages (right). The red circle shows the starting point, i.e., $r = 1$.

Compared to the other on-policy DRL algorithms, PPO usually takes a longer time to converge, however, PPO has shown a good efficiency and stability on high-dimensional continuous control problems [11]. Besides, PPO is considered easier to code than many other DRL algorithms.

The PPO algorithm [11] is defined as

Algorithm 1: PPO algorithm

for *iteration=1,2,...* **do**
  for *actor=1,2,...,N* **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end
  Optimize surrogate L w.r.t θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end

2.4 Problem definition

In this project, the objective is to learn a locomotion skill for a quadrupedal robot that is capable of walking on a rugged terrain with obstacles. The specification of the environment:

1. The terrain has a small perturbation, but is overall flat
2. There are 4 obstacles evenly placed on the terrain
3. the obstacle may be a pitfall (gap) or a stump (box)
4. the obstacles are all possible to be conquered, given an optimal agent

The specification of the experiments: an experiment is considered as a *success* if

1. The agent travels from start to the end of the terrain, and
2. with a total reward higher than the reward threshold, and
3. the agent body has never hit the ground or obstacles (felt down)

Otherwise, the experiment is considered as a *failure*. Note, the tests are evaluating on three criterion: (I) the success rate (II) the failure rate and, (III) the falling rate. Note, the success and failure rates always sum to 1 and the falling rate is a sub-category of the failure rate.

A well generalised and robust locomotion policy should have:

1. A high success rate in different terrain
2. A good strategy for overcoming the obstacles
3. A stable velocity
4. An optimised and efficient gait

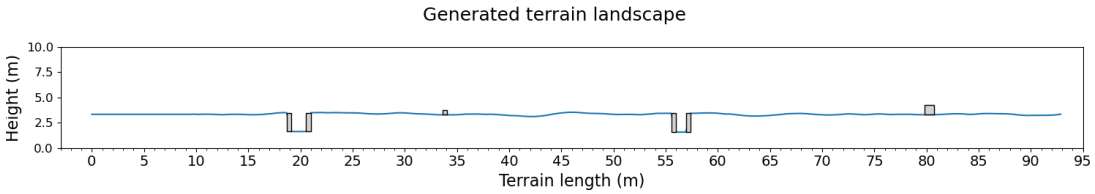
Chapter 3

Methodologies

3.1 Environment design

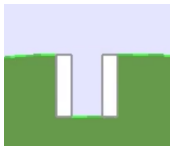
The experiments are conducted in a simulated environment, which is modified based on the *OpenAI Gym BipedalWalker environment in 2-D*. The terrain is generated according to the specification in 2.4, Figure 3.1 illustrates an example of the rendered terrain without obstacles. The simulation environment is a vertical slice of the 3-D world, with the gravity pointing at the negative direction of the y-axis. The green area is the solid ground and the blue-grey area is the sky. There may by some randomly generated clouds in the sky, which is merely a decoration without any physics property.

Figure 3.1: A rough terrain without obstacles

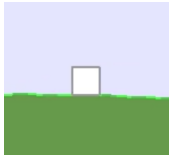


(a) A generated terrain with obstacles

(b) A rendered terrain with obstacles (different one from (a))



(c) A pitfall



(d) A stump

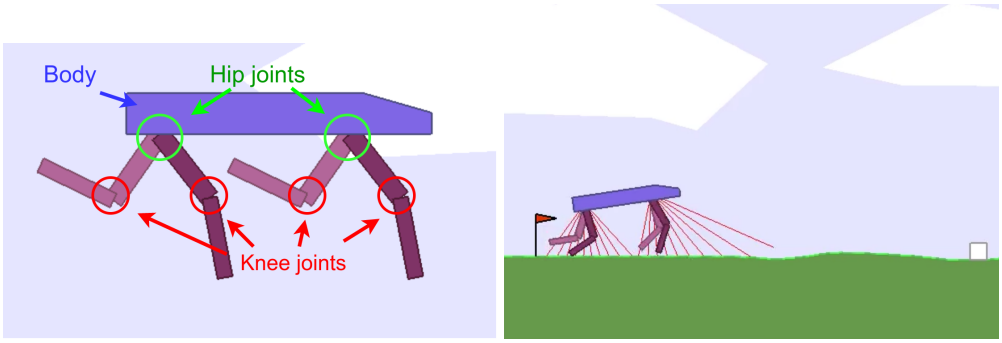
Figure 3.2: Example of the rugged terrain and obstacles

In a rough terrain, the total length is about 93 m and the ground is added with random perturbation, as shown in the figure 3.2a, where the obstacles are coloured in grey. Figure 3.2b shows a rendered terrain with obstacles. The length of the pitfall and the size of the stumps are randomised within a reasonable range. Examples of a pitfall and a stump are shown in the figures 3.2d and 3.2c. Each time the environment is initialised or reset, the terrain is randomly re-generated.

The collision boundaries are exactly as the lines and rectangles shown in the figure 3.2a. All of the terrain components are fixed in position. The gravity simulation is also simulated in the environment. An agent will be initialized at the "starting area" above the ground, which is the flat area on right of the flag (shown in the figure 3.3b). The agent is allowed to jump as high as it could, but never crossing the terrain surface or through any obstacle. On the ground, there is a line with evenly spaced segments, with an alternating colour in green and dark green. That is a distance marker, with each segment represents a distance of 1 m. The "forward direction" is defined as the positive direction of the x-axis.

3.2 Agent design

Explain more on the mass, joints and LIDAR location (variable range).



(a) The agent (b) The agent in the environment

Figure 3.3: The agent in the simulation environment

The agent is designed as shown in the 3.3a, which has 8 DoF: 4 hip joints and 4 knee joints. The perception of the agent is a 44-D vector, consists of:

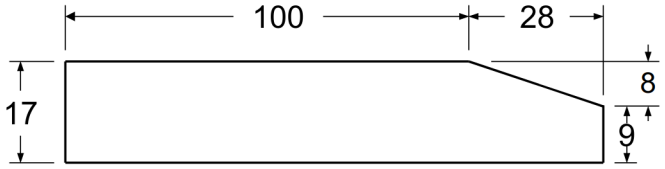
- 2-D body rotational displacement (body angle) and angular velocity
- 2-D body Cartesian velocity
- 16-D joint displacements and velocities,
- 4-D binary feet contact signal against the terrain, and,
- 20-D LIDAR (vision) at front/rare body within ranges (the red lines projected from the body to the ground, shown in 3.3b)

The design of each component of the robot is shown in the figure 3.4 and the dimensions are in meters (m). The body of the robot (figure 3.4a) is a rectangle with a cutout

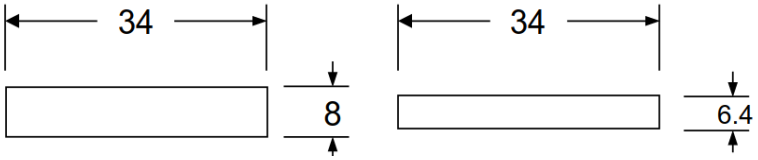
in the front. Besides, the upper leg (figure 3.4b left) and the lower leg (figure 3.4b right) are also rectangles. All together, the agent is designed as shown is the figure 3.4c, where the mounting positions of the LIDAR sensors are marked by the orange stars. The LIDAR sensors can have a different detection ranges, but is always fixed at the positions as shown.

The upper legs are attached to the body by the "hip" joints, which are 14 m and 94 m away from the left edge of the body, in figure 3.4c. The upper leg and the lower leg is connected to the "knee" joints at the center of the bottom/top of their shorter edges. The joints are shown as circles in the figure 3.4c, and the connection points are marked by the red X's in the figure 3.4d. Note that, the joints are not visible in the rendered graphics.

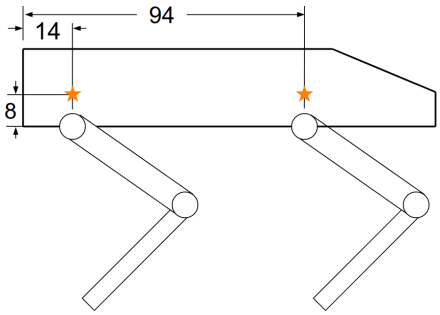
The collision only happens between the agent and the terrain components, and, there is no self-collision between the body and the legs. In addition, the legs are labeled from 0 to 3, in an order of: front left, front right, rare left and rare right. The left legs are coloured in red-grey and the right legs are coloured in dark-red, see figure 3.3a.



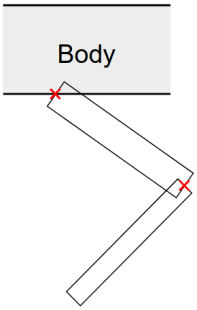
(a) The body



(b) The robot legs: upper leg (left), lower leg (right)



(c) Design of the agent



(d) The connection points

Figure 3.4: The blueprint of the agent, in meters (m)

The friction coefficient is set to 2.5 between the legs and the obstacles/terrain. This friction coefficient allows the agent to climb over the large stumps. On the other hand, it is still possible for the agent to make a slip.

In the simulation, the lengths are scaled by a factor of SCALE $= 1/30$. And after scaling, the mass of each link = area of the link $\times \frac{1}{\text{SCALE}^2} \times$ material density ρ,

- The body is 11.47 kg, where $\rho_{body} = 5$ kg/m^2.
- The upper leg is 0.30 kg, where $\rho_{upper\ leg} = 1$ kg/m^2.
- The lower leg is 0.24 kg, where $\rho_{lower\ leg} = 1$ kg/m^2.
- The total mass of the robot = $m_{body} + 4 \times (m_{upper\ leg} + m_{lower\ leg})$= 13.63 kg.

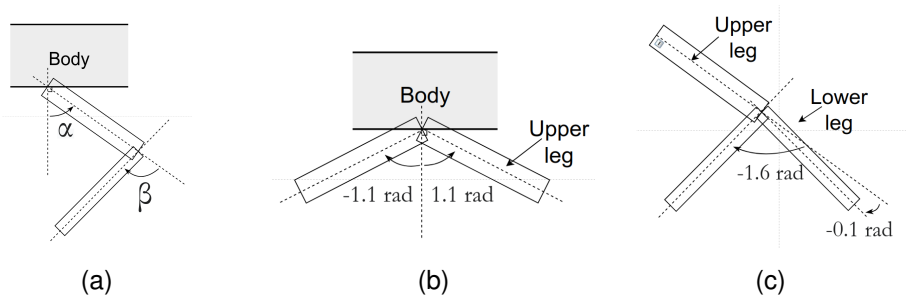


Figure 3.5: Agent joint limits

The hip and knee joint positions are defined as α and β (figure 3.5a) for each leg. And the joint limits are $\alpha \in [-0.8, 1.1]$ rad (figure 3.5b) and $\beta \in [-0.1, 1.6]$ rad (figure 3.5c).

3.3 Implementing Forward Dynamics

Due to the lack of support of the original BipedalWalker environment, a FD algorithm is required for the simulation. More specifically, the original environment has provided the dynamics for handling collisions between rigid bodies. It is only the joint-level torque control that is unsupported. According to section 2.2, the FD is implemented and embedded into the simulation environment:

Algorithm 2: Forward Dynamics algorithm

Function `GetJointVelocity`($\dot{\theta}_{curr}$, I, τ)**:**

$\ddot{\theta} \leftarrow \tau / I$

$\dot{\theta}_{new} \leftarrow \text{clip}(\dot{\theta}_{lower\ limit}, \dot{\theta}_{upper\ limit}, \ddot{\theta} \times \Delta t + \dot{\theta}_{curr})$

return $\dot{\theta}_{new}$

The algorithm takes the moment of inertia I (provided by the simulator) and the current joint velocity $\dot{\theta}_{curr}$ and return the new joint velocity $\dot{\theta}_{new}$ for the next timestep, which is clipped by the joint velocity limits. The joint velocity limits are defined as

- $[-4, 4]$ rad $\cdot$ s^{-1} for the hip joints, and
- $[-6, 6]$ rad $\cdot$ s^{-1} for the knee joints.

If assume $f_{PD} = 100$ Hz, the Δt is 0.01 s, which is small enough for simulation, so that, the error introduced by this approximation is almost trivial. Notice that, the joint velocity calculated from the function *GetJointVelocity* is the theoretical joint velocity. Together with other dynamics, such as collisions, the simulator calculates and provides the real results to the agent.

3.4 PD controller

From the industry, the integral term in the PID controller is sometimes omitted. If the desired position is unreachable, i.e. not in the C-space, the output torque will increase monotonically. This happens because after reaching the joint limit, $\mathbf{q}_e(t)$ becomes a non-zero constant, so $\int \mathbf{q}_e(t)\,dt$ increases monotonically regardless of its direction. Thus, a PD controller is used in this project.

As discussed in 1.2.1, MELA adopts 25 Hz trajectory sampling frequency for the agent that is proven to be successful. Hence, this project also adopt the 25 Hz sampling frequency. The PD controller is operating at another specific ***control frequency*** in Hz, which should be sufficiently high, and also reasonable for the computing speed.

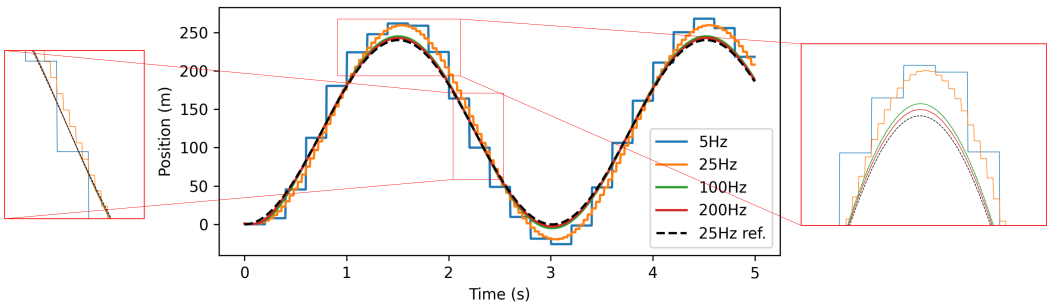


Figure 3.6: Response of 25 Hz sinusoidal input with different control frequencies (plotted in 250 Hz).

The figure 3.6 shows the response of the mass-spring-damper model (mentioned in section 2.1.4) controlled under different f_{PD}'s, a higher control frequency is shown to have a smaller error. Especially the overshooting at the crests and troughs of the trajectory, see in the zoom-in plot on the right of the figure 3.6. However, the improvement of the response becomes too small when f_{PD} is large. In conclusion, for 25 Hz trajectories, 100 Hz control frequency is sufficient.

The torque limit is $[-80, 80]$ Nm for every joint. This value is inherited from the original environment configuration that has been adopted by many other works, such as Song et al. [14]. For our agent, the "ratio of mass against the number of actuator" is about $13:8$, which is about the same as on the original agent BipedalWalker. Therefore, this torque limit allows the agent to generate sufficient power for the locomotion.

The PD gains are fine-tuned for each actuator on the agent. With a sinusoidal trajectory input, as the light blue curves shown in the figure 3.7, the top two rows are the response of each joint in the blue curves and the bottom two rows are the corresponding torques generated from the actuators.

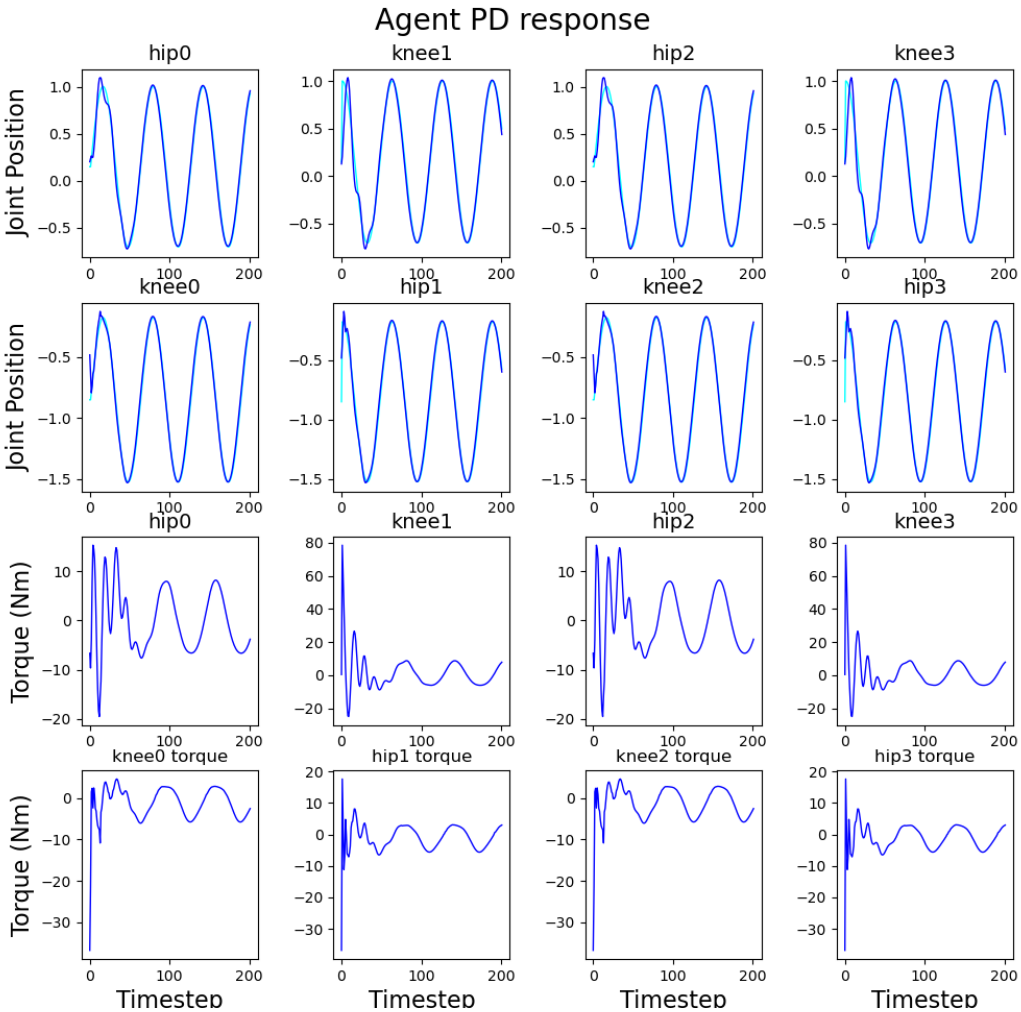


Figure 3.7: Response of the agent with 25 Hz sinusoidal input. We can see from the plots, the torque response is fluctuating drastically in the first 50 timesteps, and then becomes smooth. This is considered to be normal for accelerating the joints from the rest, besides, the joint positions are closely following the desired values.

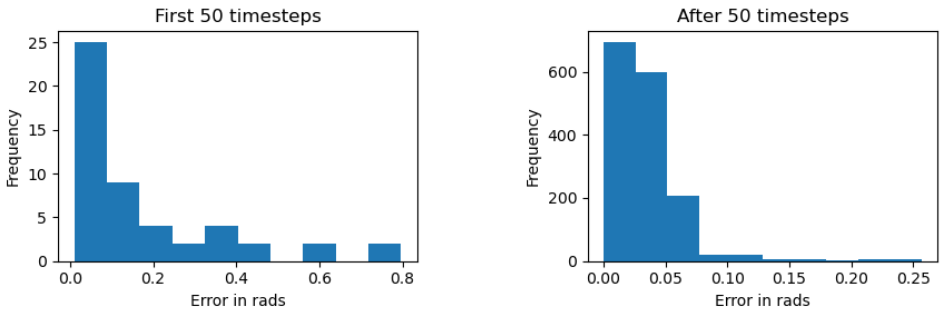


Figure 3.8: Histograms of the PD error

Together with the error histograms (3.8), in the starting phase (figure 3.8 left), although the joint error is skewed towards 0, there are occasionally large errors that are greater than 0.4 rad (22.9 degrees), such as 0.80 rad (45.8 degrees). After 50 timesteps, shown in the figure 3.8 (right), joint error is basically below 0.05 rad (2.9 degrees), with very rarely error between 0.1 to 0.25 rad (5.7 to 14.9 degrees). Under gravity, there is an inevitable steady-state error for a PD controller, however, this performance is considered accurate enough for this project.

3.5 The training methodology

3.5.1 The training algorithm

Based on the discussion of DRL and PPO in the section 2.3, the training algorithm is defined as:

Algorithm 3: The training algorithm

$iteration \leftarrow 0$
initialise $\theta \sim \mathcal{N}(\mu, \Sigma)$
while *policy entropy not converged AND iteration* $< N_{train}$ **do**
  while *sampled length* $\leq T_{train}$ **do**
    Collect the training data with policy π_θ with the rollout agents
  end
  Split the training data into fragments of length M
  for *update iteration=1,* $\ldots$ *,*N_{SGA} **do**
    Optimize surrogate L w.r.t θ with one randomly selected fragment
  end
  $\theta \leftarrow \theta_{optimised}$
  $iteration \leftarrow iteration + 1$
end

Where N_{train}, T_{train}, N_{SGD}, M are constants, μ is a zero vector and Σ is the identity matrix, where the dimension of μ and Σ varies according to the dimension of the policy network. The entropy convergence detection algorithm is defined in the algorithm 4.

The training starts with a randomly initialised policy network π_θ, collect the training data and update the policy by the SGA algorithm as mentioned in the PPO algorithm (section 2.3.2). The hyperparameters can be found in the Appendix A.

3.5.2 The rollout phase

The training data is collected in the rollout phase. The rollout process for an individual rollout worker (agent) is illustrated in the figure 3.9. In the rollout phase, many parallel rollout agents collects the training data independently, concurrently, in their simulation environments. Until enough training data is collected, the sampled data are concatenated into one big training data, which is then split into smaller fragments of length M from the training algorithm (algorithm 3).

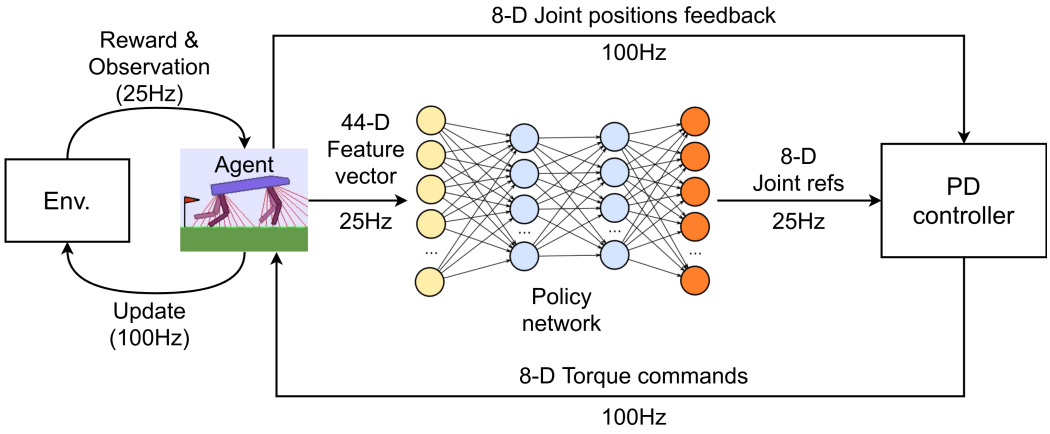


Figure 3.9: Pipeline of the rollout process, from which training rollouts are collected. The rollout agent observes from the environment, feeding the normalised observations as an input to the policy network and receives the next action (the next trajectory) to take. At the same time, the PD controller drives the agent to act in the environment according to the output trajectory. Then, the simulator updates the environment and provides a new observation to the agent. This process is repeated until the reanimation conditions are met.

Data normalisation is important in DRL. Without the normalisation, the DNN may experience many problems that prevents the DNN from learning correctly, e.g. the *vanishing gradient* and the *exploding gradient* problems [15]. Therefore, the values in the observation vector are all normalised to $[-1, 1]$.

3.5.3 Condition of early termination

Once the policy has converged, further training may still improve the gait smoothness by a little bit. But, too many iterations of extra training is a waste of time. Therefore, a concise algorithm is design for finding a good balance between the training quality and the efficiency:

Algorithm 4: Policy entropy convergence detection

Function `hasEntropyConverged`(*an array of entropy,* α)**:**
- **if** $length(\alpha) \geq S$ **then**
  - $k \leftarrow$ the gradient of the best fit line for the last S values in α
  - **if** $|k| < threshold$ **then**
    - **return** True
  - **end**
- **end**
- **return** False

Where S and threshold $\in (0, 1)$ are constant parameters. With this function, the training algorithm is able to terminate the training at a few training iterations after the policy has converged. The parameters can be found in the Appendix A.

Chapter 4

Experiments and evaluation

4.1 Introduction

The following sections discuss different methodologies with evaluation. The discussion begins with the experiment with the baseline model, which displays many issues. After that, we propose a novel methodology for overcoming those problems, with the evaluations for demonstrating its versatility and completeness.

4.2 The baseline experiment

For a recap, the DRL problem can be simplified as a maximisation problem over the objective function, which is the reward function. The reward function defines the behaviour that are rewarded and the behaviour that are penalised. As discussed in the introduction 1.2.2, the performance quality largely depends on a good the reward function. The baseline model sets the bar of the minimum quality of the task that has to be achieved.

This project is running on a brand new derivation from the OpenAI's BipedalWalker challenge, which means there is no other work or baseline available for a comparison. Hence, we introduce a "naive" experiment to serve as the baseline experiment.

4.2.1 Reward shaping

The baseline reward function is defined as:

$$\text{reward}_t = \begin{cases} \text{P}_{falling}, & \text{if the agent falls down} \\ c_1(\text{RS}_t - \text{RS}_{t-1}) - c_2\|\tau_t\|_1 - c_5|v_x - v_x^*|, & \text{otherwise} \end{cases} \tag{4.1}$$

Where.

$$\text{RS}_t = \begin{cases} c_3\mathbf{x} - c_4|\theta_{body}|, & \text{if } t \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

Where $c_1, \ldots, c_5$ and $P_{falling}$ are constants, τ is a vector of the torque command, $\mathbf{x}$ is the body displacement in x-axis, θ_{body} is the body angle and the RS is the "Robot Shaping" reward, that rewards the agent for a good movement or penalises for a bad movement.

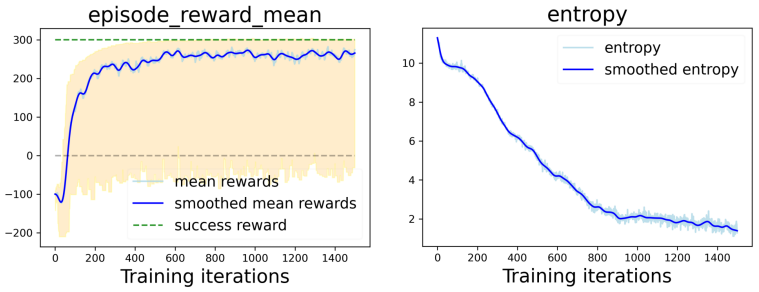
Firstly, the term $c_1(RS_t - RS_{t-1})$ is the only source of gaining positive reward. If the robot has moved forward, then $\mathbf{x}$ is positive, so there is a reward. But, if the robot has moved backwards, there is a penalty. Secondly, the torque penalty $c_2\|\tau_t\|_1$ penalise the use of torque, because an optimal policy should be more efficient, and therefore, consume less power. Thirdly, the term $-c_4|\theta_{body}|$ penalises if the body is not flat. Finally, if the agent's x-velocity differs from the target x-velocity $\mathbf{v}_x^*$, there is a penalty of $c_5|v_x - v_x^*|$. This term is added as required in section 2.4, the agent is desired to have a stable velocity.

4.2.2 Experiment results

4.2.2.1 Completeness

The training results are shown in the figure 4.1a, where the yellow shaded area is the range of the reward value, the deep blue curve is the smoothed mean reward, the light blue curve is the raw data of the mean reward, and the green dotted line the success reward threshold. Note, in the training mode, the actions are sampled stochastically from the policy network. In the testing mode, the policy network always output the mean of the distribution. This can be the reason that the mean reward has never hit the success reward threshold during training. Therefore, the mean reward from training does not reflect if the training has succeeded, or not.

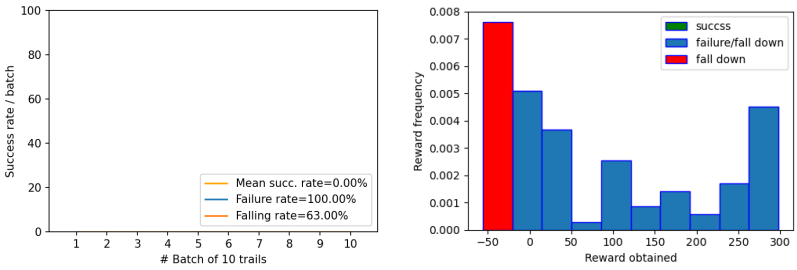
From the test results shown in figure 4.2a, a 0% success rate indicates the locomotion learning has failed. Despite of that, the training algorithm is considered to be correct. As shown in the figure 4.1b, after 900 iterations of training, the policy entropy has converged to about 2. At the same time, the mean reward has converged to about 250.



(a) The training mean reward (b) The training entropy training

Figure 4.1: Training results of the baseline model

The result from 100 testing episodes is shown in the figure 4.2a and the histogram 4.2b. From the testing results, we can see about 2/3 agents has fallen down and about 1/3 agent scored below the success reward threshold.



(a) Test results (bars are all 0%) (b) Test reward Histogram

Figure 4.2: Training results of the baseline model. (a) The failure rate is 100%, where the falling rate takes 63% of the failing rate. (b) The minimum, mean and maximum testing rewards are -56, 78.2 and 298.0, respectively. The distribution of the rewards is biased towards the lower end, but with a few high scores. None of them has passed the success reward threshold.

4.2.2.2 Conflicting velocity rewards

A major reason for causing this sub-optimal gait is that, the reward function is self-contradictory. The term $\mathrm{RS}_t - \mathrm{RS}_{t-1}$ in the reward function is unbounded, that promotes the agent for being be as fast as possible in terms of the x-velocity. More formally,

$$\text{reward}_t \propto \Delta\text{RS} = \Delta(c_3\mathbf{x} - c_4|\theta_{body}|) \tag{4.3}$$

Assume θ_{body} is always close to zero. And because Δt is small, then

$$\text{reward}_t \propto \delta\mathbf{x} = v_x \tag{4.4}$$

On the other hand, the term $\mathrm{P}_{v_x} = -c_5|\mathbf{v}_x - \mathbf{v}_x^*|$ is limiting the x-velocity to be steady, therefore, there is an contradiction.

4.2.2.3 Ability of overcoming obstacles

Another issue is overcoming the obstacles. The model hasn't learned an optimised policy to jump over the stumps. As shown in the snapshots[1], the agent is tripped over by a small stump, which should be very easy for a well learned policy to overcome.

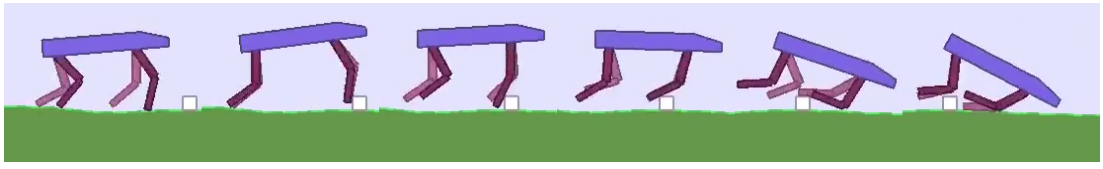


Figure 4.3: The baseline agent fails at a small stump (duration: 1s)

Similar to the stumps, the agent constantly gets stuck at the pitfalls. See the snapshots[2] shown in the figure 4.4.

[1]The video is at "attached videos/01 agent tripped by a small stump/"
[2]The video is at "attached videos/02 agent stuck in a pitfall/"

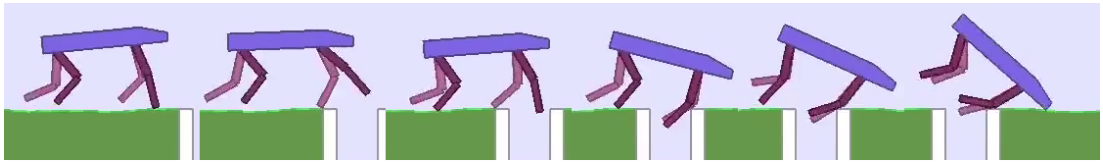
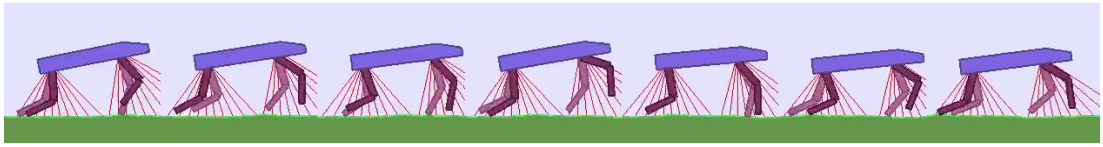


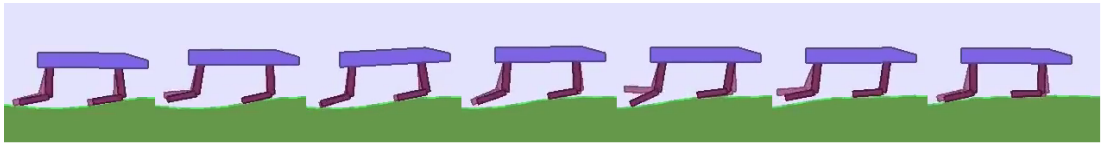
Figure 4.4: The baseline agent stuck in a pitfall (duration: 1s)

4.2.2.4 Torque smoothness

Overall, the baseline policy is more intend to use the front legs for driving the agent forward. We can see the front legs gait displays a better "walking" gait while the rare legs are more like they are being dragged along (see the video[3]).



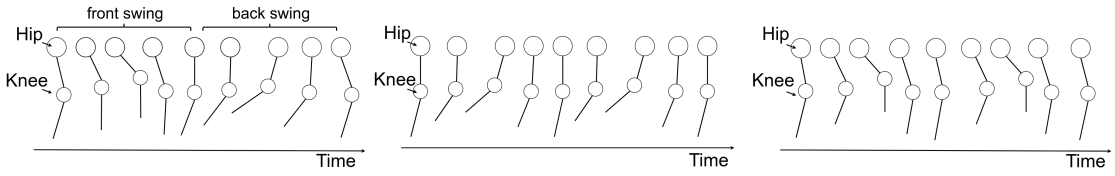
(a) The skipping agent (at about 4 m/s)



(b) The crawling agent (at < 0.5 m/s)

Figure 4.5: Demonstration of the bad gaits

Figure 4.7 shows a very noisy joint trajectories, especially the knee joints. As for the torque commands, shown in the figure 4.8), they are also very noisy. There are a lot of sharp or irregular torques commands. Particularly, it seems that the knee of the leg 1 and 2 are less utilised, compared to the knee of leg 0 and 3. The above reasons are causing the legs to shake and being inefficient. From testing, the shaky torque command usually results in two types of gaits: skipping[4] (figure 4.5a) and crawling[5] (figure 4.5b). Where the in a skipping gait the front legs only performs a half swing: back swings only or front swings only (figures 4.6b and 4.6c).



(a) A full swing (b) Back swings only (c) Front swings only

Figure 4.6: Examples of the different leg swings

[3]The video is at "attached videos/02 agent stuck in a pitfall/"
[4]The video is at "attached videos/03 skipping/"
[5]The video is at "attached videos/04 crawling/"

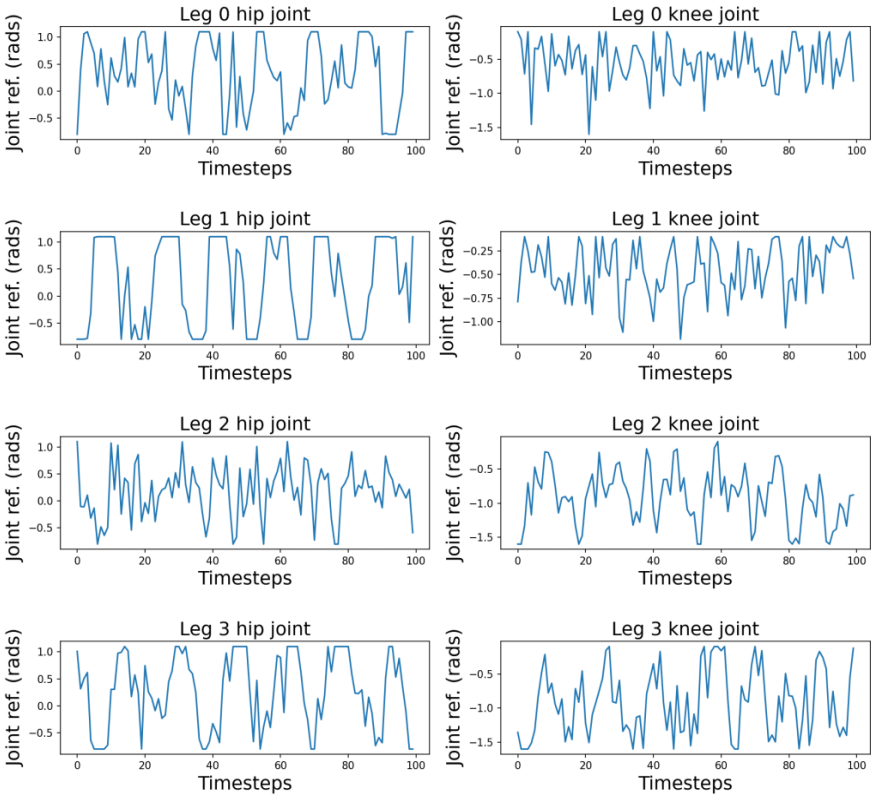


Figure 4.7: Example of baseline torque commands in 25 Hz

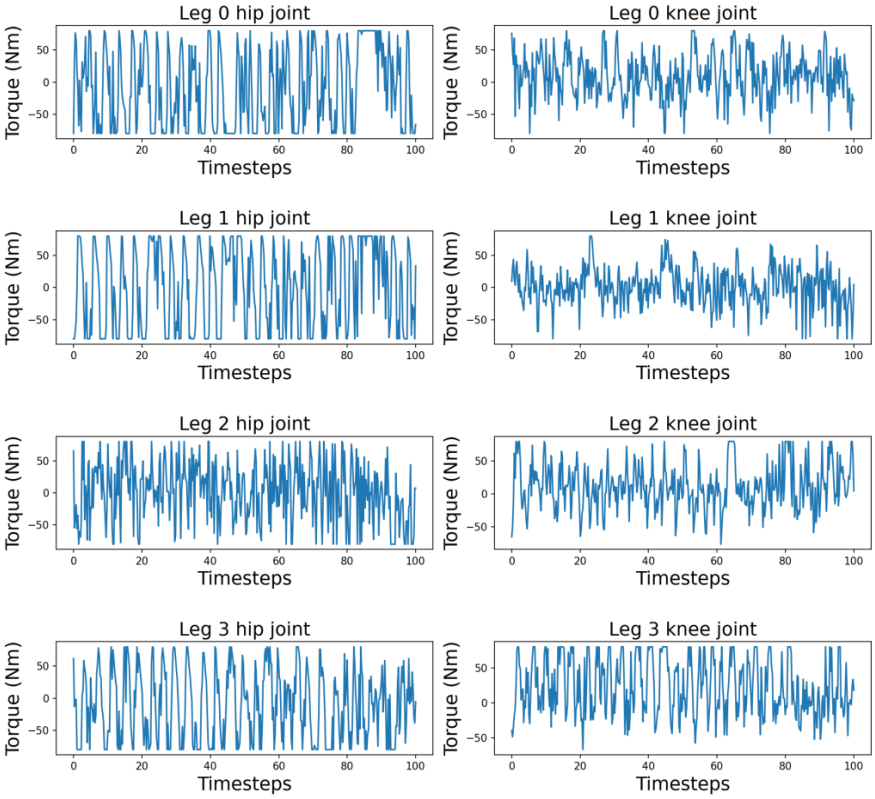


Figure 4.8: Example of baseline torque commands in 100 Hz

4.2.3 Experiment conclusion

Although the baseline experiment is a failure, it is still a good proof of concept to the project. In the same time, it has revealed many issues which are useful to the later discussions. The policy has learnt to go forward, but in a poor gait. Overall, it has shown a great potential of active obstacle avoidance and a more stable travelling velocity.

4.3 The novel approach

This section introduces a novel design of the reward function, the *Dynamic Reward Strategy*, known as DRS. Many experiments are discussed for justifying the design of each component, and for evaluating the performance.

Firstly, according to the problem definition, a smooth locomotion with a steady velocity is preferred. Therefore, the new reward function removes the "robot shaping" reward, function RS, from the baseline reward function. The new reward function rewards the agent when its velocity is in a desired range at the target x-velocity v_x^*. The choice of limiting the x-velocity, not the 2-D velocity vector is discussed in section 4.3.2. Secondly, to eliminate the shaky gait, DRS penalises the square of the joint angular acceleration, which is discussed in section 4.3.3. Finally, DRS introduces a conditional penalty strategy on the joint torque and angular acceleration to further boost the locomotion performance. This is discussed in the section 4.3.5.

4.3.1 The new reward shaping

The new reward function removes the self-contradicting terms form the baseline reward function while adding some new terms, which is defined as:

$$\text{reward}_t = \begin{cases} \mathrm{P}_{falling}, & \text{if has fallen down} \\ c_1(c_2 - c_3(v_x - v_x^*)^2) + \|\mathbf{k}_1 \circ \boldsymbol{\tau}\|_1 + \|\mathbf{k}_2 \circ \ddot{\mathbf{q}}\|_2^2, & \text{otherwise} \end{cases} \tag{4.5}$$

Where, $c_1, \ldots, c_3$ and $\mathrm{P}_{falling}$ are constants, $\boldsymbol{\tau}$ is the torque command and $\dot{\mathbf{q}}$ is the joint angular acceleration. The symbol "$\circ$" denotes the element-wise vector multiplication. Coefficients $\mathbf{k}_1$ and $\mathbf{k}_2$ are coefficient vectors that are further discussed in section 4.3.5.

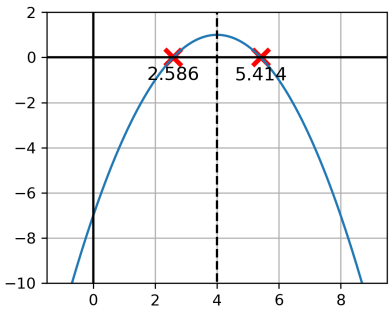


Figure 4.9: The velocity reward shaping (with $v_x^* = 4.0$ m/s)

Further speaking the velocity reward, the reward shape is shown in the figure 4.9. In the figure, the desired velocity is v^*. The tolerance thresholds are v^*_{lower} and v^*_{upper}, which are cross points (2.586 m/s and 5.141 m/s) of the curve at the x-axis. Within the thresholds, there is a positive reward that is inversely proportional to the squared velocity error. Beyond the thresholds, i.e. if the agent moves too slow or too fast, there is a penalty proportional to the squared velocity error.

4.3.2 Target velocity

Three experiment are trained in the rough terrain (rough means the perturbation is included, in contrast to the flat terrain) with the new reward function but different target velocities. Tests are conducted in both rough and flat terrain, A well generalised agent is expected to perform equally well in either terrain, where the flat terrain is never seen by the agents during training.

Shown in the table 4.1. The experiment 3 is considered a failure, as the success rates are both 0%. Because, the target velocity of $[5,0]^\top$ m/s is too high which is impossible for the agent to maintain at, so, the agent has never scored a reward over the threshold. From experiment 2, the agent scored a success rate about 80% in the both terrain, that is 9% and 23% higher than the corresponding success rates from the experiment 1. However, the second agent is less stable, as in each terrain, the falling rates are about 10% higher than the first agent. The target y-velocity is set to 0 m/s, as specified in the section 2.4, the terrain is overall flat and there is no demand on the vertical displacement. For the moment, $[4,0]$ m/s is a better target velocity.

For experiment 1 and 2, The skipping and crawling gaits are both considered bad, which is further discussed in the section 4.3.3. Besides, the first two experiments are neither a robust or generalised locomotion model, because they are sensitive to the terrain perturbation and do not perform as good as in an easier flat terrain.

Exp.	Gait	Target Vel.	Rough terrain		Flat terrain	
			succ. rate	falling rate	succ. rate	falling rate
1	crawling	$[2,0]^T$	70%	2%	65%	0%
2	skipping	$[4,0]^T$	79%	13%	88%	8%
3	trotting	$[5,0]^T$	0%	20%	0%	18%

Falling is a sub-category of failure, and the failure rate = 1 - success rate

Table 4.1: Target velocity evaluation (from 100 testing episodes in each terrain)

Further analysis with the experiment 2, the agent often gets stopped by large stumps, even if it has never felt down. For an example, See the snapshots[6] 4.10. Together with many other rollouts, the agent lacks of the jumping ability.

[6]The video is at "attached videos/05 agent failed overcoming a stump/"

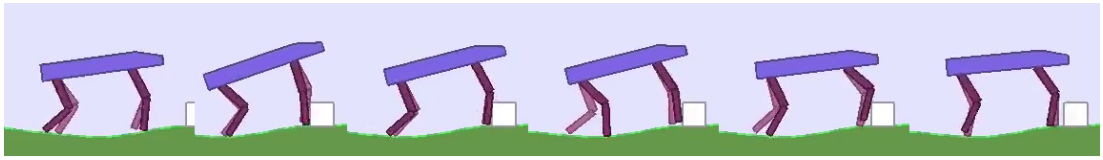


Figure 4.10: The agent from experiment 2 is being stopped by a large stump, but, it has never felt down, until the maximum episode length is hit, the episode has terminated. (duration: 20s)

Inevitably, the jumping motion introduces large y-velocities during the leap, which it is heavily penalised by the reward function with the y-velocity constrained. In the same time, the body has to tilt in order to jump. Therefore, for learning a better jump gait, the y-velocity and the body must not be penalised. Those penalty terms will be removed in the final reward function design. Combining with the experiment results, $v_x = 4$ m/s is an appropriate target velocity. This is further justified in section 4.5.

4.3.3 Joint angular acceleration penalty

As mentioned in section 4.2.2.4, the joint trajectories are not smooth. This "shaky" behaviour can be formalised as a high frequency direction change of the joint velocities $\Delta\dot{\mathbf{q}}$ which is the integration of the joint acceleration

$$\Delta\dot{\mathbf{q}} = \dot{\mathbf{q}}_t - \int_t^{t+\delta t} \ddot{\mathbf{q}}\, dt \tag{4.6}$$

Then,

$$\Delta\dot{\mathbf{q}} \propto \int_t^{t+\delta t} \ddot{\mathbf{q}}\, dt \tag{4.7}$$

Therefore, an instant penalty is applied to the joint angular acceleration, at each time step. In the end, this penalty is accumulated. The square is introduced to further penalise the large joint accelerations. In additionally, the Euclidean norm is applied, because each joint is penalised equally. The joint angular acceleration penalty:

$$\text{penalty}_t \propto \|\ddot{\mathbf{q}}\|_2^2 \tag{4.8}$$

With this static penalty strategy, an example of the joint trajectories and the torque commands are shown in the figures 4.11 and 4.12. The leg 1 is not included in the discussion below, as it is used for balancing and providing extra versatility (further discussed later). The joint trajectories are very smooth, compared to the baseline. The periodic trajectories enables a more natural swing motion of the legs. The torque commands are neat and well optimised. Besides, the controller generates basically the full range of the torque while displaying a fluent torque transition without drastic changes.

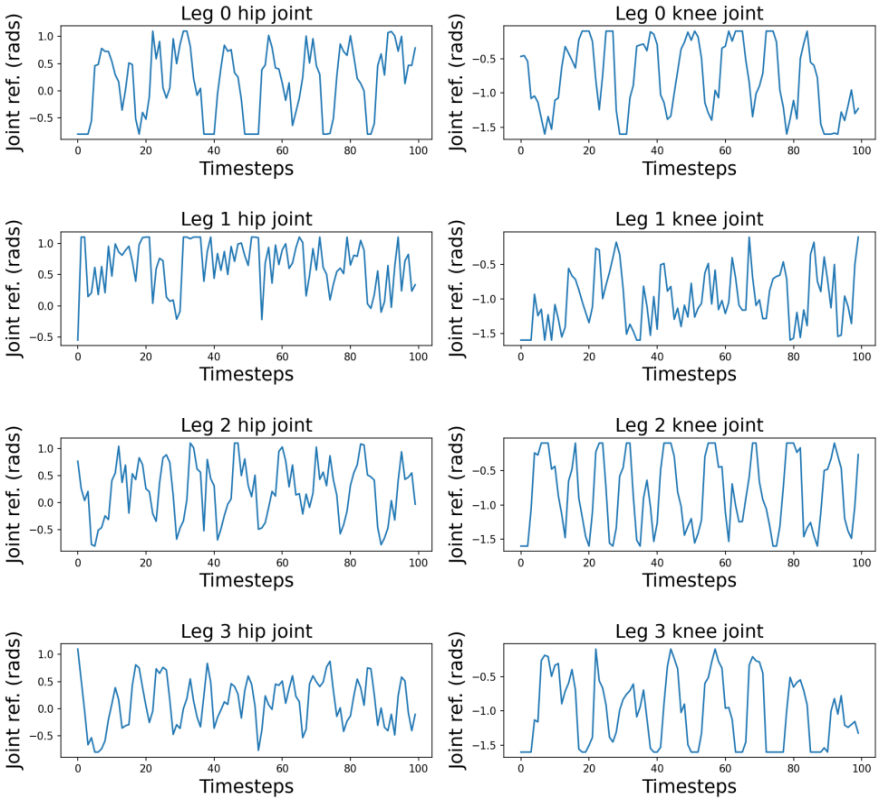


Figure 4.11: Example of the 25 Hz joint trajectories with the angular acceleration penalty

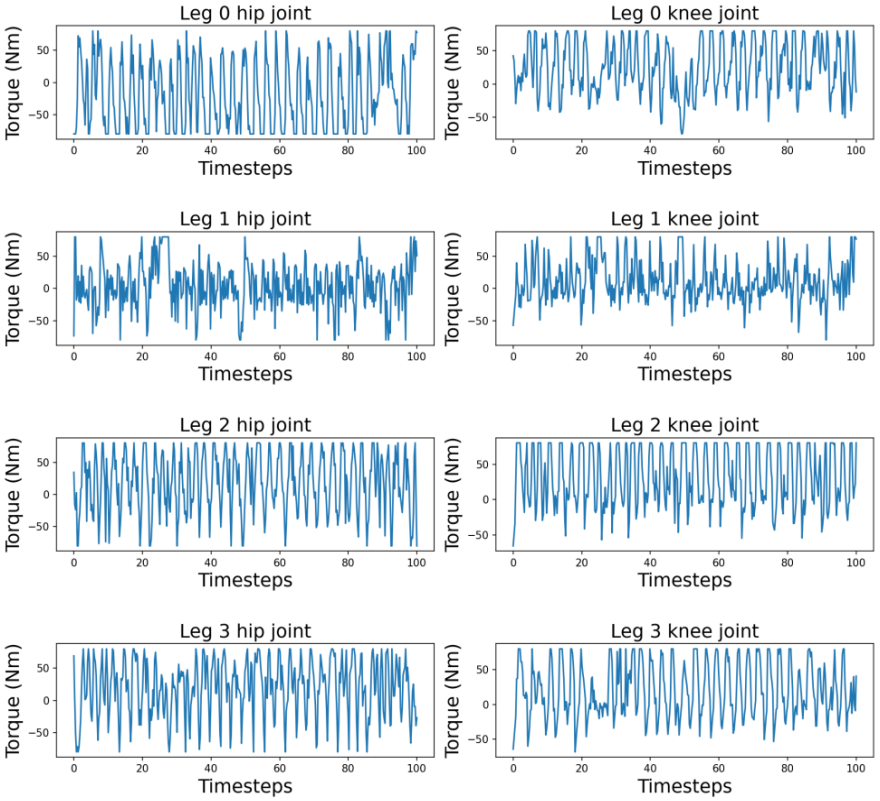
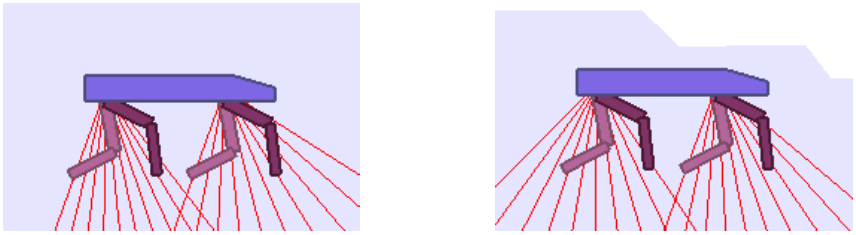


Figure 4.12: Example of the 100 Hz torque commands with the angular acceleration penalty

4.3.4 A better vision perception

If the agent detects an obstacle too soon, from observation, the agent usually jumps earlier. Hence, there is a higher chance that the front legs will jump into a pitfall or get tripped over at a stump. To address this, the range of the front and rare LIDAR sensor is adjusted. As shown in the figure, the new LIDAR configuration is front and rare LIDAR is sensing at ranges $[-20, 50]$ degrees (front) and $[-45, 34]$ degrees (rare), as shown in the figure 4.13b, whereas the old ranges are $[0, 58]$ degrees (front) and $[-20, 58]$ degrees (rare), shown in the figure 4.13a). In the new LIDAR configuration, the agent has a shorter range in the front, but a bigger range at the back. The ground below the body is always well-covered by either configuration.



(a) The old LIDAR configuration (b) The new LIDAR configuration

Figure 4.13: Modification of the robot vision sensors

Exp.	LIDAR conf.	Target Vel.	Terrain with perturb.		Without perturb.	
			succ. rate	falling rate	succ. rate	falling rate
2	old conf.	$[4, 0]^T$	79%	13%	88%	8%
4	new conf.	$[4, 0]^T$	85%	14%	94%	5%

Falling is a sub-category of failure, and the failure rate = 1 - success rate

Table 4.2: LIDAR configuration evaluation (from 100 testing episodes in each terrain)

Here are two experiment (table 4.2) comparing the performance between the old and new LIDAR configurations, while other settings are kept the same. As shown in the table, there is an 6% success rate improvement, in both terrains, after the LIDAR configuration update. Despite the new model is still sensitive to the perturbations, the new agent, from experiment 4, has achieved 85% success rate in the rough terrain and a surprising 94% success rate in the flat terrain, which is impressive.

4.3.5 The Dynamic Reward Strategy

After the above experiments and evaluations, the model is still having a falling rate at $\approx 15\%$ in the rough terrain. From the sampled episodes, the falling is mainly caused by a slip step.

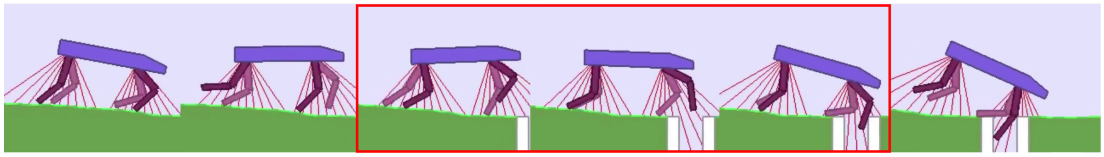


Figure 4.14: The agent falls into a pitfall by a slip

Here is an example of a slip[7] (figure 4.14), the leg 0 has slipped off the ground, which then, caused a tilt in the body and a misstep of leg 0 (the moment is highlighted by the red box). Finally, the agent gets stuck in the pitfall and the experiment has failed.

The behaviour of the slip is summaries as following:

1. The legs are shaky and swings too slow in the air.
2. The lower legs take too long to land. Therefore, the agent misses good footholds.
3. The contact force is too small. Therefore, it does not support the body firmly.

Together with the new reward function, DRS is defined as:

Algorithm 5: Dynamic Reward Strategy

```
for each leg in the quadrupedal robot do
    if the lower leg is in contact then
        Decrease the joint torque and joint angular acceleration penalty coefficient
    else
        /* if the lower leg is not in contact                    */
        Increase the joint torque and joint angular acceleration penalty coefficient
    end
end
```

Note that, the lower legs can only be in contact with the terrain and the obstacles. DRS promotes a high power output when the leg is in contact, and reduces the waste of energy of the legs, during a swing in the air. Further speaking, this strategy guides the policy to learn following properties:

1. Smooth swings of the leg for better foot placements and
2. Fast leg swings in the air, as the angular acceleration is more consistent, so as the joint velocity.
3. A better contact force to reduce slips and
4. A better support/acceleration of the agent, as a burst of torque is generated when the lower leg is in contact.

DRS also reduces the swing time of the legs between the steps, while performing stronger pushes in each step. In additional, fast joint velocity is allowed, because

[7] The video is at “attached videos/06 agent slips into a stump/”

the angular velocity is not penalised. Also, as discussed in the Forward Dynamics section 2.2, the joint acceleration is proportional to the applied torque, thus, the torque command is also penalised in a close relation to the joint angular acceleration.

The implementation of this strategy is done by manipulating the penalty coefficients in the reward function, such that, a different penalty is issued under different conditions. For a recap, $\|\mathbf{k}_1 \circ \boldsymbol{\tau}\|_1$ is the joint torque penalty and $\|\mathbf{k}_2 \circ \ddot{\mathbf{q}}\|_2^2$ is the joint angular velocity penalty in the new reward function. The vectors $\boldsymbol{\tau}$, $\mathbf{q}$, $\mathbf{k}_1$ and $\mathbf{k}_2$ are vectors of $\mathbb{R}^N$, where N is the number of the joints. The coefficient vectors are defined as:

$$\mathbf{k}_1 = k_{1,\, i\in[1\,\ldots\,N]} = \begin{cases} c_4, & \text{if the related foot is in contact, for leg } i \\ c_5, & \text{otherwise} \end{cases} \tag{4.9}$$

$$\mathbf{k}_2 = k_{2,\, j\in[1\,\ldots\,N]} = \begin{cases} c_6, & \text{if the related foot is in contact, for leg } j \\ c_7, & \text{otherwise} \end{cases} \tag{4.10}$$

Where $c_4, \ldots, c_7$ are constants and N, as mentioned, is the number of joints. According to the DRS algorithm defined in 5, the constants $c_4, \ldots, c_7$ have a relation of $c_4 \leq c_5$ and $c_6 \leq c_7$.

4.4 Hyperparameter tuning

This project adopts an implementation of the *Population Based Training (PBT)* algorithm [16] by Ray Tune[8] for tuning the DRL hyperparameters. The PBT algorithm combines the *Bayesian optimisation* and the *Grid search* technique to optimise the hyperparameters efficiently while training. As in the figure 4.15, each curve is one conducted experiment. The optimal hyperparameters are attached in the appendix A. Note, the reward is scaled up by a factor of 5 in this experiment.

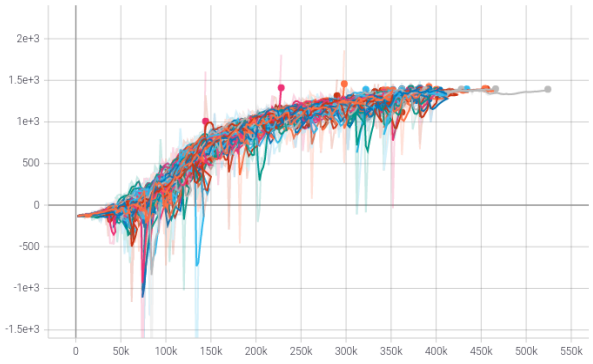


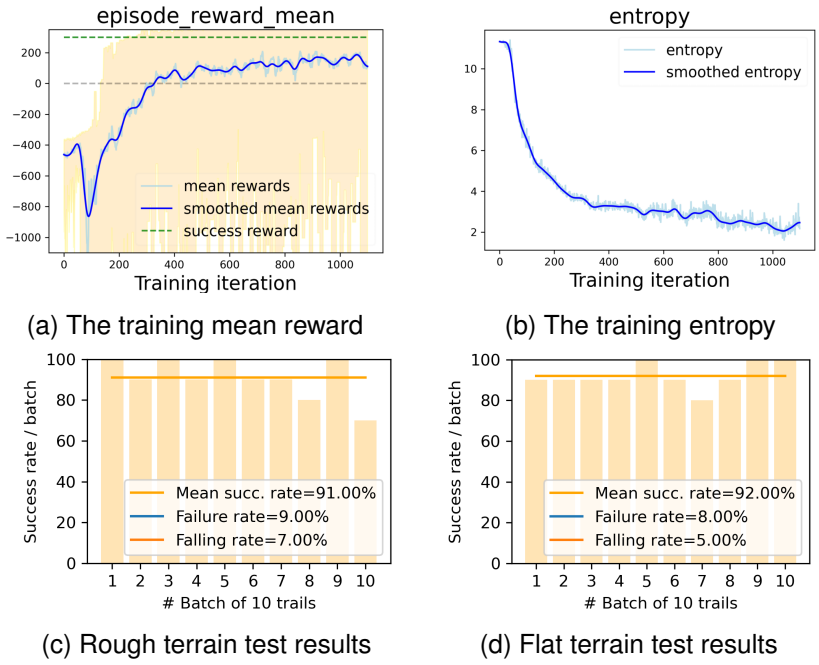
Figure 4.15: The tuning results with Ray Tune.

[8] A scalable Python library for experiment execution and hyperparameter tuning. Website: `https://docs.ray.io/en/latest/tune/index.html`

4.5 Evaluation

The hyperparameters used in the DRS (experiment 6) are attached in the Appendix A. The policy has converged after 1000 iterations of training. See the mean reward and the entropy of training in the figure 4.16a and 4.16b. The mean reward does not reflect if the training has succeeded or not, as mentioned in the baseline (section 4.2.2.1).

From 100 testing episodes, DRS has scored a 91% and 92% success rate in the rough terrain and the flat terrain, respectively. Shown in the figure 4.16 (Each yellow bar represents the success rate of a batch of 10 tests). We can see that, **the model is performing equally well with/without perturbation. Therefore, it is well generalised locomotion model.**



(a) The training mean reward

(b) The training entropy

(c) Rough terrain test results

(d) Flat terrain test results

Figure 4.16: Training & Test results of the novel approach

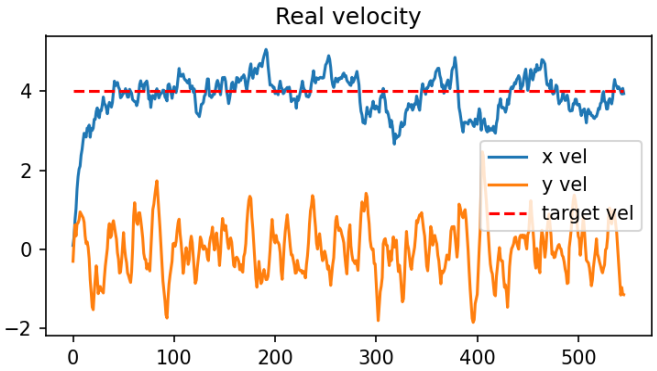
As shown in the table 4.3, **this novel approach (Exp. 6) has score the best success rates across the experiments.** Compared to the baseline model, the DRS approach has about 90% improvement of the success rates in both terrains. In the rough terrain, there is a 6% to 21% improvement in the success rate improvement compared to the others. In the flat terrain, DRS is 2% lower than success rate of the experiment 4, but is more than 10% higher than the success rates of the experiment 1,2 and 5. (The experiment 3 is ignored, as it is a failure)

Another important improvement is about the falling rate, compared to the other approaches with a high target velocity, but without the dynamic penalty (experiment 2-5), DRS has scored the least falling rate of 7% and 5% in the rough terrain and the flat terrain. Especially compared to the experiment 5, with the only difference of the dynamic penalty, DRS yields an about 3.5 times reduction on the falling rate. **This indicates that, DRS is having a very positive impact on the stability of the locomotion model.**

Exp.	LIDAR	Target Vel.	Penalty type	Gait	Rough terrain		Flat terrain	
					succ. rate	falling rate	succ. rate	falling rate
baseline	old	$[2,0]^T$	static	failed	0%	0%	N/A	N/A
1	old	$[2,0]^T$	static	crawling	70%	2%	65%	0%
2	old	$[4,0]^T$	static	skipping	79%	13%	88%	8%
3	old	$[5,0]^T$	static	trotting	0%	19%	0%	17%
4	new	$[4,0]^T$	static	skipping	85%	14%	94%	5%
5	new	$v_x^* = 4$	static	trotting	72%	25%	83%	17%
6	new	$v_x^* = 4$	DRS	trotting	91%	7%	92%	5%

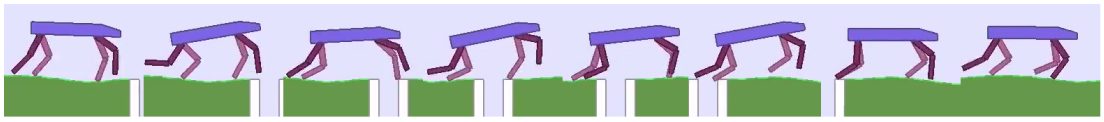
Falling is a sub-category of failure, and the failure rate = 1 - success rate

Table 4.3: The final evaluation (from 100 episodes for each terrain)

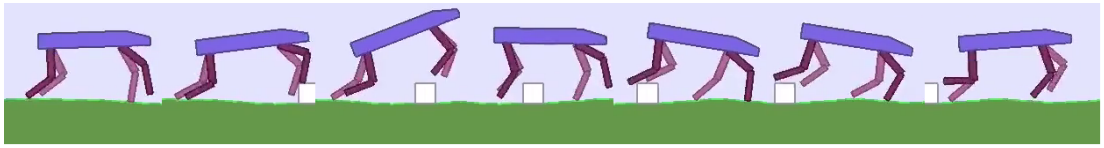


(a) The velocity curve of this sample

(b) The trotting gait of the agent



(c) The agent jumps over a pitfall



(d) The agent jumps over a stump

Figure 4.17: Evaluation of DRS

As expected, the gait style of DRS displays all the desired properties as discussed in the section 4.3.5. See the snapshots[9] shown in the figure, the agents performs a trotting gait where the legs swing fast and smooth 4.17b. Even the leg 4 is less utilised, it provides extra versatility when it is critically needed. This has largely improved the robustness of the locomotion. At a pitfall, the agent overcomes it easily, shown in the snapshots 4.17c. At a stump, the agent performs a good jump, as in the snapshots 4.17d. In addition, shown in the figure 4.17a, the x-velocity is very stable at 4 m/s, unless at a stump. **Overall, DRS displays an optimal locomotion policy of overcoming the obstacles.**

More about the trajectories, with DRS, the joint trajectories and the torque commands are at the same level as the policy with a static penalty strategy, which are considered smooth and optimal. Shown in the figures 4.18 and 4.19. **Combining all from of the above, DRS learns a versatile and robust locomotion for the quadrupedal robot.**

[9]The video is at “attached videos/07-09 novel approach/”

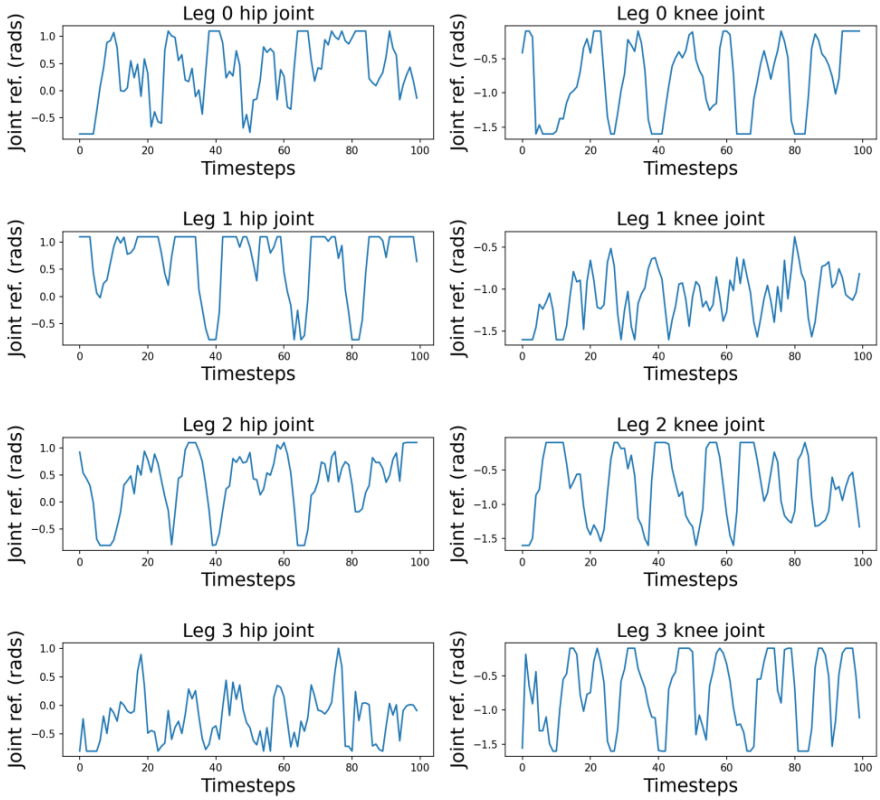


Figure 4.18: Example of the 25Hz joint trajectories of DRS

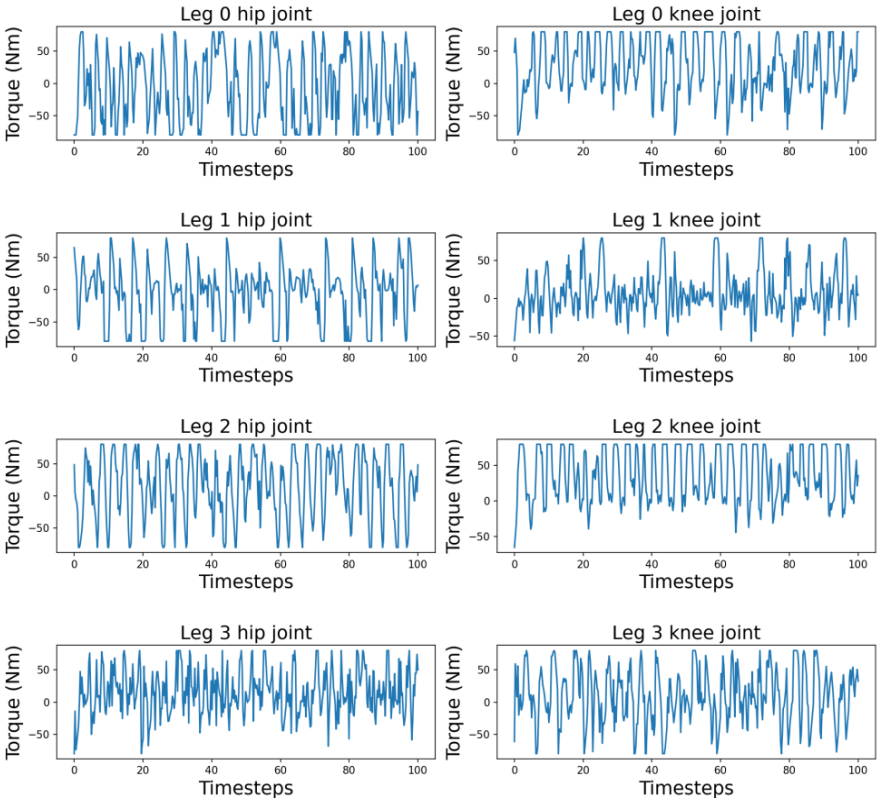


Figure 4.19: Example of the 100Hz torque commands of DRS

Chapter 5

Conclusion and discussions

5.1 Conclusion

This project proposes a novel approach, the *Dynamic Reward Strategy (DRS)*, for learning a variety locomotion skills on the quadrupedal robot. The learned policy requires only the sensory and vision inputs that is not depended on any form of the history data. This approach has been proven to be robust and versatile in the rugged terrain with random obstacles. The learned model has shown its redundancy of skill in 2-D, such that, a good locomotion is performed with just three legs, while the fourth leg comes to rescue only when the agent is out of balance.

The low-level feedback control is considered a success. Under the appropriate configuration, the PD controller fundamentally provides a good manoeuvrability to the quadrupedal robot. Experiments and evaluations have shown that, the PD controller generates a correct and smooth torque through out the project.

At last, the training time of DRS is considered fast, where the model usually converges within 2 hours of training using just one CPU with 3 cores at 4.0 Ghz. Although it is not directly comparable, but for a reference, other approaches with the quadrupedal robot in 3-D usually takes about 3 to 10 hours to converge, which also require more CPUs and GPUs.

5.2 Limitation

The major limitation of this project is the dimensionality. It is considerably easier to learn a locomotion model in 2-D, than learning in 3-D. Because in 3-D, the C-space and the workspace of the robot becomes much more complicated and there are many more issues to consider, such as the orientation of the body, the diagonal/sideway gait, steering, foothold planning and more.

5.3 Future works

A future work can be generalising this model to the 3-D world. There are so many potential applications of the legged robot locomotion, such as autonomous delivering, disaster rescue and more. Also in 3-D, more locomotion skills could be learned under the framework of DRS, such as walk along a pre-defined path, climb up stairs and etc. With the vision input in 3-D, this approach could also learn to behave like a guide dog, which would cost significantly less time and money than training a real guide dog (training for up to 2 years, $59,600 in total[1]). This would hugely benefit more blind people.

Inspired by MELA [4], another direction of the future work can be learning a locomotion with dynamic velocities. The goal is to learn a self-adapting locomotion that actively adjust its gait and velocity according to the user command and the environment. For example, if walking on the ice where the friction is low, the agent automatically slows down its velocity and starts to crawling. When climbing up a slope, the agent knows to incline its body forward. Anyway, there is unlimited potential in the field of the legged robot locomotion.

[1] Source: `https://puppyintraining.com/how-much-does-a-guide-dog-cost/`

Bibliography

[1] Navvab Kashiri, Andy Abate, Sabrina J. Abram, Alin Albu-Schaffer, Patrick J. Clary, Monica Daley, Salman Faraji, Raphael Furnemont, Manolo Garabini, Hartmut Geyer, Alena M. Grabowski, Jonathan Hurst, Jorn Malzahn, Glenn Mathijssen, David Remy, Wesley Roozing, Mohammad Shahbazi, Surabhi N. Simha, Jae-Bok Song, Nils Smit-Anseeuw, Stefano Stramigioli, Bram Vanderborght, Yevgeniy Yesilevskiy, and Nikos Tsagarakis. An overview on principles for energy efficient robot locomotion. *Frontiers in Robotics and AI*, 5:129, 2018. ISSN 2296-9144. doi: 10.3389/frobt.2018.00129.

[2] Marc Raibert, Kevin Blankespoor, Gabriel Nelson, and Rob Playter. Bigdog, the rough-terrain quadruped robot. *IFAC Proceedings Volumes*, 41(2):10822–10825, 2008. ISSN 1474-6670. doi: https://doi.org/10.3182/20080706-5-KR-1001.01833. 17th IFAC World Congress.

[3] Tobias Klamt, Max Schwarz, Christian Lenz, Lorenzo Baccelliere, Domenico Buongiorno, Torben Cichon, Antonio DiGuardo, David Droeschel, Massimiliano Gabardi, Malgorzata Kamedula, Navvab Kashiri, Arturo Laurenzi, Daniele Leonardis, Luca Muratore, Dmytro Pavlichenko, Arul S. Periyasamy, Diego Rodriguez, Massimiliano Solazzi, Antonio Frisoli, Michael Gustmann, Jürgen Roßmann, Uwe Süss, Nikos G. Tsagarakis, and Sven Behnke. Remote mobile manipulation with the centauro robot: Full-body telepresence and autonomous operator assistance. *Journal of Field Robotics*, 37(5):889–919, 2020. doi: https://doi.org/10.1002/rob.21895.

[4] Chuanyu Yang, Kai Yuan, Qiuguo Zhu, Wanming Yu, and Zhibin Li. Multi-expert learning of adaptive legged locomotion. 2020. doi: 10.1126/scirobotics.abb2174.

[5] Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. 2020. doi: 10.1126/scirobotics.abc5986.

[6] Kevin Lynch. *Modern robotics : mechanics, planning, and control*. Cambridge University Press, Cambridge, United Kingdom New York, NY, 2017. ISBN 9781107156302.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driess-

che, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

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

[9] Richard Sutton. *Reinforcement learning : an introduction*. The MIT Press, Cambridge, Massachusetts London, England, 2018. ISBN 978-0262039246.

[10] Matthew Hausknecht and Peter Stone. On-policy vs. off-policy updates for deep reinforcement learning. In *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI Workshop*, July 2016.

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

[12] David Saad. *On-line learning in neural networks*. Cambridge University Press, Cambridge England New York, 1998. ISBN 978-0-521-65263-6.

[13] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[14] Doo Re Song, Chuanyu Yang, Christopher McGreavy, and Zhibin Li. Recurrent deterministic policy gradient method for bipedal locomotion on rough terrain challenge. 2017. doi: 10.1109/ICARCV.2018.8581309.

[15] George Philipp, Dawn Song, and Jaime G. Carbonell. The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions, 2017.

[16] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks, 2017.

Appendix A

Hyperparameters used in DRS

A.1 Hyperparameters for training the policy network

MLP shape: [256, 256]
MLP activation function: $tanh()$
learning rate: 0.00005
lambda: 0.95
gamma: 0.99
value function loss coefficient: 0.62
entropy coefficient: 0.00045
clip parameter: 0.3
kl coefficient: 1.0
kl target: 0.01
value function clip parameter: 10.0
stochastic gradient descent (ascent) minibatch size: 20
stochastic gradient descent (ascent) iteration: 20
train batch size: 4000
sample fragment length: 200

A.2 Parameters of the function *hasEntropyConverged*

Parameter	Value
S	500
threshold	0.025

Appendix B

Parameters of the agent

B.1 Parameters of the reward function

Parameter	Value
$P_{falling}$	-300
v_x^*	4.0
c_1	1.0
c_2	1.0
c_3	0.5
c_4	0.00096
c_5	0.00024
c_6	0.0024
c_7	0.0012

Table B.1: Table of parameters for the novel reward function

B.2 Proportional-Derivative parameters for the joint-level PD controller

	Leg 0		**Leg 1**		**Leg 2**		**Leg 3**	
	Hip	Knee	Hip	Knee	Hip	Knee	Hip	Knee
K_p (Nm/rad)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
K_d (Nms/rad)	2.0	1.0	2.0	1.0	2.0	1.0	2.0	1.0

Table B.2: Table of the Proportional-Derivative parameters for the joint-level PD controller