Exploring dynamic locomotion of a quadruped robot: a study of reinforcement learning for the ANYmal robot.

Branislav Pilňan

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

In this work I modify the HalfCheetah environment from OpenAI Gym to resemble the ANYmal robot, creating a HalfANYmal environment. I provide a brief overview of the current reinforcement learning algorithms and benchmarks to inform my choice of reinforcement learning algorithm for this work. To improve the performance on HalfCheetah, I explore and compare two ways of implementing PD control in MuJoCo. One is done entirely in MuJoCo's MJCF model definition file but does not include gravity compensation. The other one does include it but requires changes to client code.

After performing initial tests on the HalfCheetah environment, I apply the PPO algorithm to the HalfANYmal environment and perform extensive reward shaping. I also develop methods to evaluate the desirability of different reward functions by monitoring their individual components throughout training. Finally, I demonstrate that reinforcement learning can learn dynamic locomotion over both smooth and rugged terrains in the HalfANYmal environment. Surpsisingly, I discover that training on smooth surface can achieve better results on rugged terrain than training on the rugged terrain. I propose some specific further research into this discovery.

Technical note: I recommend viewing this document in an electronic rather than paper form as it includes clickable links to videos, and coloured plots.

Acknowledgements

I would like to thank Dr Zhibin Li, my supervisor, for his guidance and support without which I would not have been able to complete this work.

Table of Contents

Chapter 1

Introduction

As demonstrated by the famous videos from Boston Dynamics[1], it is possible to control robots and achieve complex and robust locomotion using analytic and deterministic approaches. Such techniques have also been researched by Wang et al. [2019] using the EBot robot shown in figure 1.1. However, these techniques often come with assumptions about the environment which might not always hold in the real world. [Wang et al., 2019] Considering the recent advances in Machine Learning (ML), and the fact that robot control is an optimisation problem (what series of local actuations leads to the desired global outcome the fastest or cheapest?), it makes sense to try to apply the ML techniques to robot control. Reinforcement Learning (RL) is the area of ML which does this.

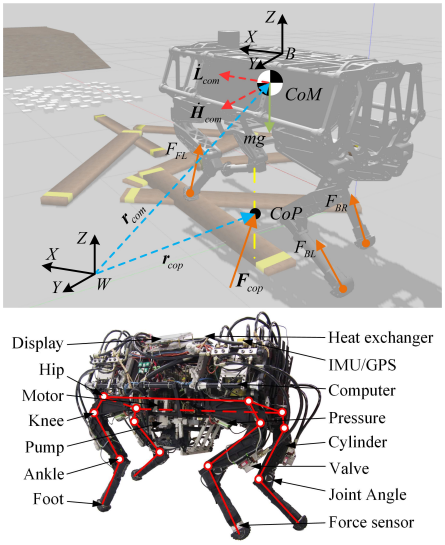

Figure 1.1: EBot robot. Pictures are from [Wang et al., 2019], provided by Dr. Zhibin Li.

One issue with reinforcement learning is that it learns by trial and error and often needs to make a lot of errors before it becomes useful. This is especially problematic with physical robots where the errors can lead to physical damage and increasing the speed of gathering experience is not as simple as using a more powerful computer. This might be why simulators seem to be very popular in reinforcement learning research (e.g. Schulman et al. [2017], Duan et al. [2016], Brockman et al. [2016], and Tassa et al. [2018] all use simulators).

There are whole suites of completely artificial simulated robots meant for reinforcement learning research. Brockman et al. [2016] and Tassa et al. [2018] describe two of them. However, these do not directly solve the problems of training reinforcement

[1]For example `https://youtu.be/rVlhMGQgDkY` or `https://youtu.be/fUyU3lKzoio`

learning algorithms on physical robots since there are no physical robots corresponding to the simulation and thus any learned behaviours cannot be directly applied to an actual robot. Although there will always be at least a small gap, between simulated and real robot, some reinforcement learning research is being done on realistic simulations of physical robots. For example, Yang et al. [2018] used a complex and accurate simulation of a real humanoid robot.

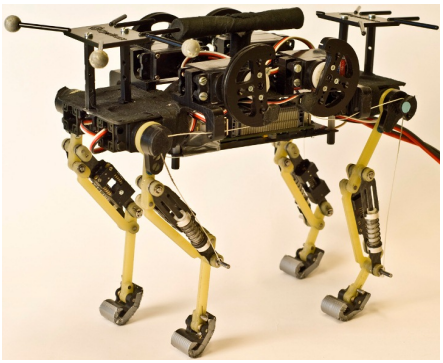

Figure 1.2: Cheetah-Cub robot.

As discussed by Raibert et al. [2008], reliable dynamic control for legged robots is potentially of great practical value since humans and animals can get through a much greater variety of terrains than wheeled or tracked vehicles can. The hope is that legged robots can replicate this performance. For this reason, Raibert et al. developed the BigDog robot which can be seen in action in this video[2] by Boston Dynamics. BigDog is similar to EBot shown in figure 1.1. They are both large robots suitable for carrying heavy payloads over harsh terrains which can be useful for example for military or rescue operations. On the other end of the spectrum are tiny quadrupedal robots like Cheetah-Cub which is only about the size of a small house cat. Shown in figure 1.2[3], Cheetah-Cub is meant primarily as an inexpensive research platform [Spröwitz et al., 2013].

Somewhere in the middle on the size scale is the ANYmal robot (shown in figure 1.3)[4] with is about the size of an average dog. According to Hutter et al. [2016], ANYmal has been designed to be fully autonomous, unlike BigDog, which is remote-controlled. ANYmal is best suited for inspection of buildings or industrial sites after accidents or natural disasters when it is unsafe for humans to enter [Hutter et al., 2016]. Such use calls for a very versatile and robust control system and is unlikely to be able to accommodate the assumptions that usually come with analytic control approaches. This makes ANYmal a particularly attractive robot to apply reinforcement learning techniques to.

Figure 1.3: ANYmal robot is very versatile.

[2] https://youtu.be/cNZPRsrwumQ

[3] Picture source: https://biorob.epfl.ch/wp-content/uploads/2019/02/CheetahCubMarkersSmall.jpg

[4] Picture source: https://www.anybotics.com/wp-content/uploads/2018/04/anymal_b_elevator-300x300.jpg

Due to aforementioned limitations of RL, I will use the MuJoCo simulator [Todorov et al., 2012] to create a planar (2D) version of the ANYmal robot [Hutter et al., 2016]. Then I shall demonstrate that my creation can learn to walk or run using reinforcement learning. Specifically, the goals of this project are:

- Modify an existing simulation to match the ANYmal robot;
- Use reinforcement learning to achieve stable locomotion on a flat surface in the modified simulation;
- Use reinforcement learning to negotiate rugged terrain in the modified simulation.

Once this proof of concept works, it will serve as a foundation for further research which will try to apply my techniques to the real ANYmal robot in the Edinburgh Centre for Robotics. (The local availability of the ANYmal robot was also an important reason for choosing it for the experiments.)

1.1 Original contributions

In this project, I have used existing implementations of reinforcement learning algorithms from OpenAI and extensively modified one of the environment implementations from their Gym framework. I have also developed custom code to simplify execution and evaluation of experiments. I have designed and carried out many experiments and analysed their results. All my original contributions can be summarised as follows:

- Designing and evaluating reward functions for HalfCheetah and HalfANYmal robots;
- Designing methods for evaluating desirability of reward functions;
- Improving efficiency of an existing python implementation of Butterworth filter;
- Adding filtering to HalfCheetah and HalfANYmal environments;
- Implementing MuJoCo HalfANYmal model and corresponding OpenAI Gym environment;
- Implementing PD control and gravity compensation for the simulations;
- Reviewing the current literature on reinforcement learning algorithms.

Chapter 2

Background and Related Work

This chapter provides some basic information about Reinforcement Learning, including the terminology used in this report. It also explains why I did not use the DeepMind Control Suite as was the original aim of the project and presents the alternatives that I chose.

2.1 Reinforcement Learning

This section describes some basic intuitions behind reinforcement learning and defines the terms *agent*, *environment*, *reward function*, *episode* and *policy* . Later, I compare the results of different reinforcement learning algorithms from the literature to choose one to use to train the ANYmal robot.

Sutton and Barto [2018] describe reinforcement learning (RL) as learning, through trial and error, what to do in which situation so as to maximise a numerical reward. The two basic entities in a common reinforcement learning setup are the *agent* and the *environment*. Figure 2.1 shows their interaction. The interaction happens in discretised time steps. In each time step, the agent observes the state $S_t \in \mathcal{S}$ of the environment and chooses an action $A_t \in \mathcal{A}$ (i.e. it decides what to do in the given situation), where $\mathcal{S}$ is the set of all possible states called the state space and $\mathcal{A}$ is the set of all possible actions called the action space. Both sets can be infinite. The selected action is then communicated to the environment which updates its state and calculates the reward

$$R_{t+1} = r(S_t, A_t, S_{t+1}) \tag{2.1}$$

where $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function*. The agent uses the reward to learn whether it did well or not. [Sutton and Barto, 2018]

The time steps are usually organised into larger units called *episodes*. Each episode starts with an initial state S_0 and ends in a terminal state S_T. The terminal state can be determined by some condition (e.g. the agent reached the goal or failed at its task), in which case the episodes are usually variable in length (T). Alternatively, an episode can be terminated after some fixed number of time steps. [Sutton and Barto, 2018]

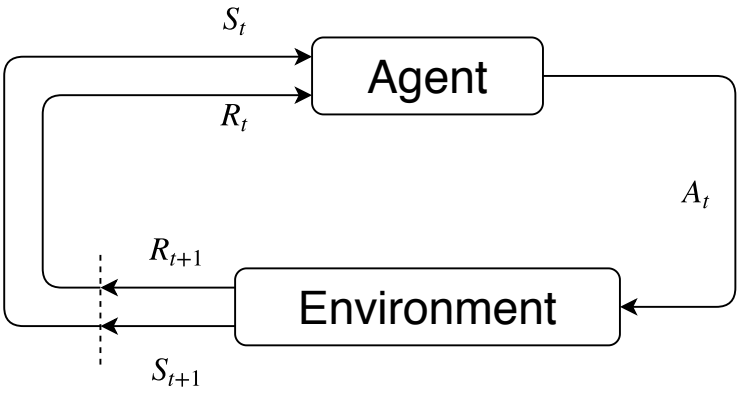

Figure 2.1: Agent-environment interaction in reinforcement learning

To make decisions, the agent uses a *policy*. Policy is a probability distribution over the action space given the current state, usually denoted as

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{2.2}$$

for any time step t. Fundamentally, reinforcement learning algorithms try to optimise this probability distribution to maximise the expected sum of rewards in an episode. Intuitively, we can say that the algorithms try to make the actions that lead to higher rewards more likely under π and the actions that lead to lower rewards less likely. [Sutton and Barto, 2018]

2.1.1 Algorithms

The aim of this project is to develop new simulated robot, similar to the ANYmal robot and demonstrate that it is possible to use RL to control the robot in the simulation. As explained in the subsequent sections of this chapter, the simulation that I use as a starting point does not come with an implementation of an RL algorithm that works well for it so I needed to find one. I set the following criteria for a good algorithm:

- It is necessary that the algorithm has been demonstrated to achieve good performance on the environment that I use as a starting point, and that this performance is reproducible (i.e. the optimal hyperparameters are available).
- There should be a reliable implementation of the algorithm that I can access and use so that I can save time and focus on developing and debugging the ANYmal environment.
- The algorithm should not have too high computational requirements, i.e. it should be possible to train an agent on a CPU in less than 24 hours and using less than 8 GB of RAM. (These are roughly the resources that I have available.)
- The algorithm should be relatively simple to understand and implement in case I needed to do so (for example if there was a problem with the implementation or if I wanted to experiment with changing something).

- If possible, the algorithm should not be too sensitive to hyperparameter settings so as to maximise the chance of it working unmodified on the ANYmal environment.

Schulman et al. [2017] present the Proximal Policy Optimisation (PPO) technique and compare it to other algorithms, namely Cross-Entropy Method (CEM), Trust Region Policy Optimisation (TRPO), Advantage Actor Critic (A2C) and vanilla Policy Gradients (PG). On the HalfCheetah environment which I use (see section 2.4), PPO performed much better than all the other algorithms in terms of the achieved reward. A2C was the second best, followed by PG, TRPO, and finally CEM performed the worst [Schulman et al., 2017]. Duan et al. [2016] benchmarked eight RL algorithms on several environments including HalfCheetah. The algorithms included CEM, TRPO, and several other algorithms not covered by Schulman et al.. Duan et al. confirmed that TRPO works better than CEM and the only algorithm to beat TRPO was Deep Deterministic Policy Gradient (DDPG). Tassa et al. [2018] compared three algorithms on a similar Cheetah environment (discussed in section 2.2) and DDPG achieved the best result. The other two algorithms (A3C and D4PG) were not included in any of the previously mentioned comparisons. In summary, PPO and DDPG showed the best performance out of contemporary RL algorithms but there does not seem to be any comparison between the two in current literature.

Both DDPG and PPO have open-source implementations by OpenAI (see section 2.3) which includes tested hyperparameter settings. However, preliminary testing showed that DDPG had much higher computational requirements[1]. On a CPU, DDPG executed almost 4 times slower than PPO and its memory requirements grew with the increasing amount of training, eventually reaching well over 8GB. PPO, on the other hand, only consumed small and constant amount of memory (less than 1GB).

More importantly, DDPG is more complex and according to Lillicrap et al. [2016], it does not provide any convergence guarantees meaning that it can be unstable. On the other hand, Schulman et al. [2017] claim that PPO is stable, reliable, and simple to implement, "requiring only few lines of code change to a vanilla policy gradient implementation."

Based on these findings, I use the PPO algorithm throughout this project.

2.2 DeepMind Control Suite

The DeepMind Control Suite is a collection of well-tested continuous control environments for benchmarking continuous RL algorithms [Tassa et al., 2018]. It includes the Cheetah environment (shown in figure 2.2) which is a planar biped robot similar to the target ANYmal robot.

Originally, my first task was to retrain the model that DeepMind used for the Cheetah

[1] More careful analysis later revealed that this is not as bad as I first thought and DDPG might still fit into my compute budget.

Figure 2.2: The Cheetah environment from DeepMind Control Suite. (Pictures generated by me, using DeepMind's open-source code. [DeepMind Technologies, 2019])

environment to achieve the result in this video[2]. This way I would get a feel for the compute requirements of training a model which would allow for better planning of experiments and I would also establish a baseline which would help me see if any of my changes to the simulation have any effect on performance. However, after inspecting the DeepMind Control Suite's source code [DeepMind Technologies, 2019], I discovered that no learning algorithm implementation was provided, rendering DeepMind's results hard to reproduce.

2.3 OpenAI Baselines

OpenAI Baselines is an open-source "set of high-quality implementations of reinforcement learning algorithms" [Dhariwal et al., 2017] which I decided to use. It includes the PPO algorithm.

The main advantage of using OpenAI Baselines is reliability. Since the algorithm implementations come from a reputable company and are checked by the open-source community, they are unlikely to include any bugs. Together with OpenAI Gym, the baselines package offers a full and tested solution where I can start changing one thing at a time to observe the effects.

The close integration of OpenAI Baselines with OpenAI Gym is also a disadvantage because it makes the algorithm implementations from the baselines package very difficult to use with tasks not within the OpenAI Gym framework (e.g. the ones from DeepMind Control Suite). The implementations of the algorithms are also difficult to understand as they use TensorFlow and are not very well separated from data logging and interfacing with the task implementations.

However, it is simple to modify the OpenAI Gym environments or define new ones which is the main aim of this project. Since I am not attempting to modify or optimise any RL algorithms for this project, the combination of OpenAI Gym and OpenAI Baselines constitutes the perfect platform for carrying out the project.

[2] https://youtu.be/rAai4QzcYbs?t=58

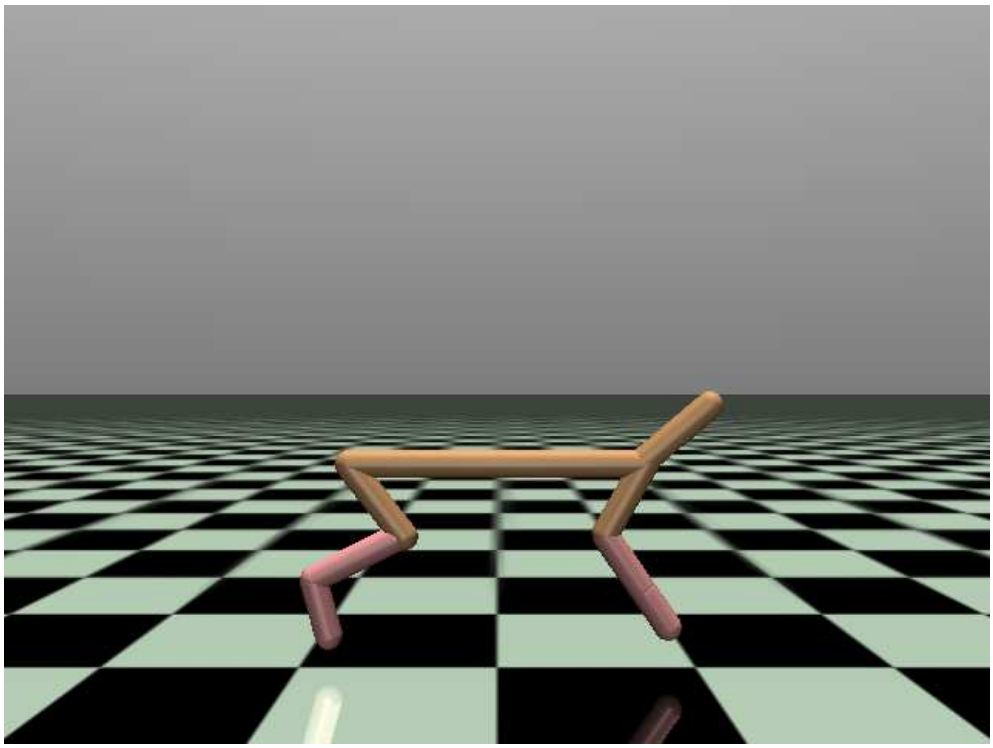

Figure 2.3: The HalfCheetah environment from OpenAI Gym. (Picture generated by me, using OpenAI's open-source code. [OpenAI, 2018])

2.4 OpenAI Gym

Similar to DeepMind Control Suite, OpenAI Gym is a collection of environments for benchmarking RL algorithms [Brockman et al., 2016]. However, there are two crucial differences. Firstly, OpenAI Gym does not focus exclusively on continuous control tasks but also includes other kinds of environments (e.g. Atari games). Secondly, the reward design is less standardised across different environments which makes it more difficult to interpret the learning curves.

Most importantly, OpenAI Gym includes the HalfCheetah environment (shown in figure 2.3) which uses the same physics as the Cheetah environment from DeepMind Control Suite and only differs in some visual aspects and has a slightly different reward function. I chose this environment over DeepMind's Cheetah as a starting point because it works with OpenAI Baselines "out of the box."

Chapter 3

Learning Dynamic Locomotion - a study on HalfCheetah

As previously mentioned, DeepMind did not publish the learning algorithms used to achieve their results so I used an implementation by OpenAI. Before modifying the simulation to match ANYmal robot (and possibly introducing bugs), I wanted to make sure the learning algorithm works with the tested HalfCheetah environment.

The first result was slightly disappointing as the robot learned to flip on its back and kick its legs in the air to bounce forwards as can be seen in this video[1]. I tried to modify the reward function to prevent this behaviour but my first attempt was not very successful.

After consulting with my supervisor, we identified the following problems with the system:

- Both the original reward function and my modified version could be improved;
- The feedback signals from the simulation to the learning algorithm were not filtered;
- The control inputs to the simulation were joint torques rather than joint positions.

3.1 Reward shaping

The default reward function in OpenAI Gym's implementation of the HalfCheetah environment was defined as

$$R = v - 0.1||\mathbf{a}|| \tag{3.1}$$

where **a** (action) is the input vector of joint torques.

Since the afent was flipping over, I tried adding the term $\left(\frac{2\alpha}{\pi}\right)^8$ to the reward function to penalise the agent for rotating more than $\alpha = \pm\frac{\pi}{2}$ rad from the level orientation.

[1] https://youtu.be/FDYzYn3YHrw

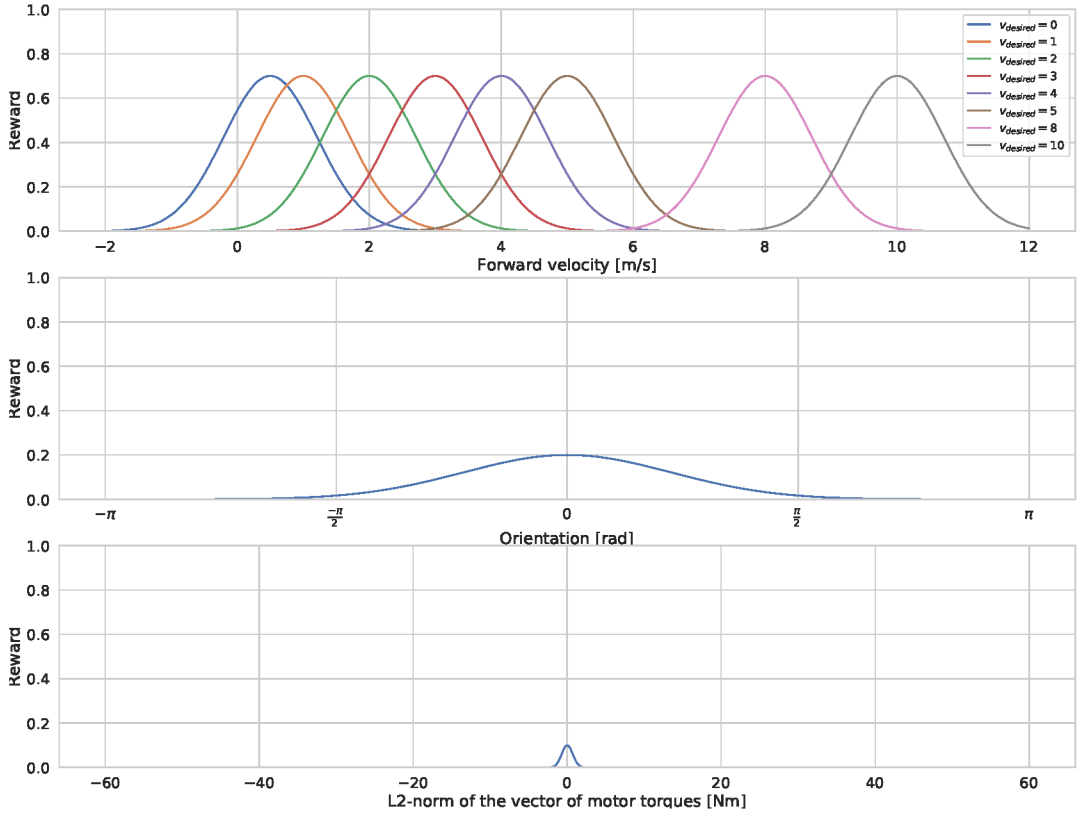

Figure 3.1: Terms of the reward function 3.2 for different values of desired forward velocity

However, this led to unstable results and large negative reward scores which were difficult to interpret.

After a discussion with my supervisor, I decided to use RBFs (Radial Basis Functions) to define "desired" values for velocity, orientation and motor torques, and designed the reward function

$$R = 0.7e^{-(v_{desired}-v)^2} + 0.2e^{-\alpha^2} + 0.1e^{-||\mathbf{t}||^2} \tag{3.2}$$

where α is the orientation of the torso (0 means the robot is level) and $\mathbf{t}$ is the vector of torques produced by each of the motors. Figure 3.1 shows the plots of the different terms of the reward function.

This design allowed for easy changes to the desired values of the individual quantities as well as to their relative importances while keeping the reward bounded between 0 and 1. It did have some problems however. The torque reward was very narrow relative to the range of possible torques which made it difficult to obtain for the agent but that was an easy fix. Bigger problem was with the velocity reward. If $v_{desired} \geq 3$, the initial state ($v = 0$) was so far out on the tail of the velocity RBF that the agent would not register any reward signal and stay stationary, collecting the angle and torque rewards only. I tried widening the velocity RBF as well and eventually I was able to achieve locomotion with $v_{desired} = 4$ for the following reward function: (visualised in

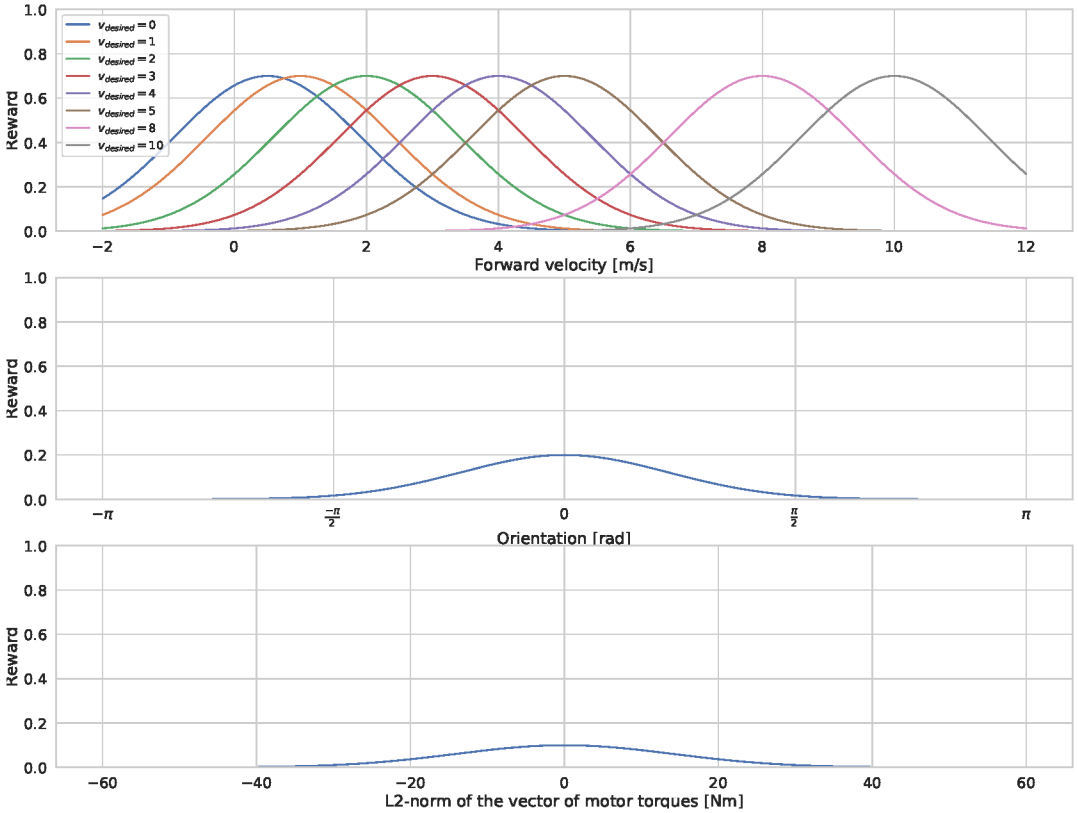

Figure 3.2: Terms of the reward function 3.3 for different values of desired forward velocity

figure 3.2)

$$R = 0.7e^{-\left(\frac{v_{desired}-v}{2}\right)^2} + 0.2e^{-\alpha^2} + 0.1e^{-\left(\frac{||t||}{40}\right)^2} \tag{3.3}$$

3.2 Filtering

I used a first-order Butterworth filter to filter the joint positions, velocities and torques which are used for calculating reward and as feedback for the learning algorithm. The implementation of the filter has been kindly provided by Chuanyu Yang, who has used it for the same purpose in [Yang et al., 2017] and [Yang et al., 2018]. Since the learning algorithm operates at 20Hz I set the cutoff frequency for the filter to the corresponding Nyquist frequency, i.e. 10Hz.

Initially, filtering slowed training down significantly since the basic implementation called for looping over all the scalars to be filtered. Since many of these scalars were already stored in numpy arrays in the OpenAI framework, I refactored the filter code to take advantage of numpy's fast array operations which restored most of the training performance.

Joint	back thigh	back shin	back foot	front thigh	front shin	front foot
k_s	240	180	120	180	120	60
k_d	6	4.5	3	4.5	3	1.5

Table 3.1: Default settings of stiffness and damping

3.3 PD control

In the default implementation by OpenAI, there was no PD control so the agent controlled motor torques but there were springs and dampers attached to each joint so the resulting torque for a joint looked something like

$$\tau(x) = x - k_s\alpha - k_d\dot{\alpha} \tag{3.4}$$

where x is the (torque) input, k_s is the spring stiffness, k_d is the damping constant, α is the joint angle, and $\dot{\alpha}$ is joint velocity. The settings of the constants are summarised in table 3.1.

This setup meant that every input corresponded to a unique and stable configuration of joint angles although the two were different numerically and the mapping from torques to positions depended slightly on the current configuration of the robot (gravity, impact forces).

To visualise how the system works, I used a dummy agent which selects 10 input torques and holds each for 10 seconds, abruptly changing to the next one at the end of each period. Figure 3.3 shows that the system does indeed achieve a stable state for each input action although there is no clear direct correspondence between the numerical value of the input torque and the resulting stable joint angle. Moreover, figure 3.3 also shows that there is significant oscillation of the joint angle after each change to torque. It is a common practise to use PD control to reduce or eliminate these effects (e.g. Yang et al. [2017], Yang et al. [2018], Li et al. [2017] and Yuan et al. [2019] all use PD control in similar settings).

To aid learning, I set out to implement proper PD control. First, I tried to do this within the MuJoCo physics model in order to minimise the changes required to python code for the environment. I implemented the PD control by attaching a position servo and a velocity servo to each of the joints so the control equation looked like

$$\tau(x) = K_p(x - \alpha) - K_d\dot{\alpha} \tag{3.5}$$

where K_p and K_d are the PD gains, x is the (position) input, α is the current position and $\dot{\alpha}$ is the velocity.

I repeated the same experiment as for the default model and used it to hand-tune[2] the PD gains. The resulting performance is shown in figure 3.4. There is almost no oscillation or overshoot and there is only a small steady-state error. Hence, this configuration is better than the deault one in OpenAI Gym which performed on all three metrics (figure 3.3).

[2]Hand-tuning is the standard practise in similar situations according to Yuan et al. [2019].

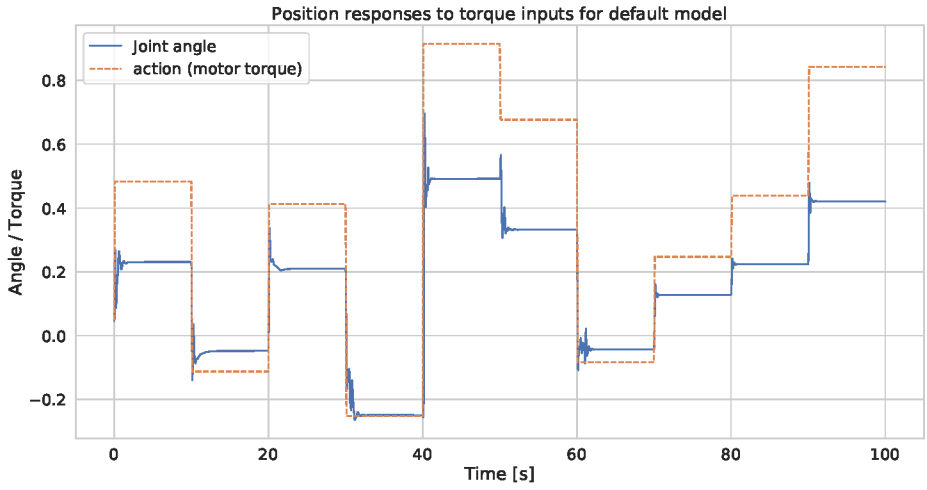

Figure 3.3: Position responses for the back foot joint. Note that the angle is limited to the range of -0.4 to 0.785.

Joint	back thigh	back shin	back foot	front thigh	front shin	front foot
K_p	1000	500	1000	1000	1000	1000
K_d	40	25	20	40	30	25

Table 3.2: PD gains

However, this method did not provide gravity compensation so the values of the PD gains (summarised in table 3.2) needed to be very big. Consequently the agent had much more force available and learned to jump very far and high as can be seen in this video[3].

3.4 Result

After implementing the new reward function and filtering, but not the PD control, the HalfCheetah agent was able to learn stable locomotion as can be seen in this video[4].

[3] https://youtu.be/o1yYmGeg44g
[4] https://youtu.be/TWJ9I08uzZI

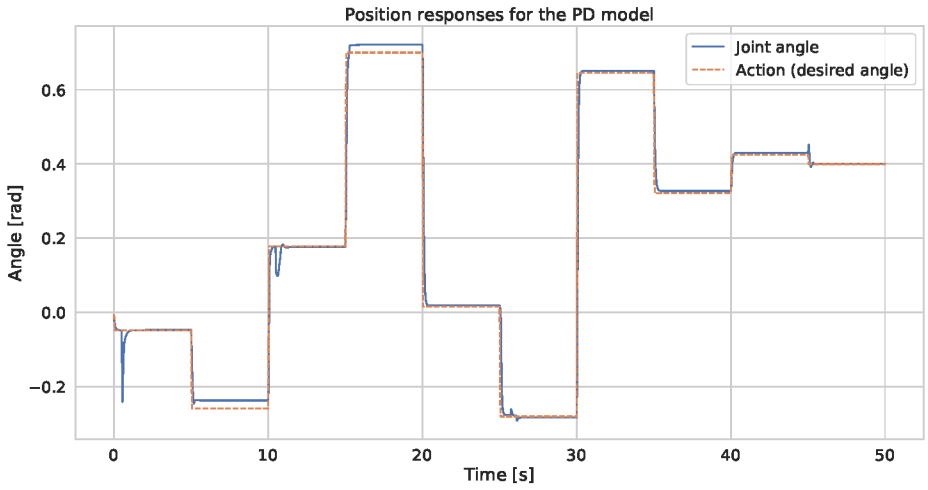

Figure 3.4: Position responses for the back foot joint.

Chapter 4

Learning Dynamic Locomotion - a study on ANYmal robot

I modified the HalfCheetah simulation to resemble the ANYmal robot (shown in figure 4.1), creating a HalfANYmal as shown in figure 4.2. The main changes include:

- Removing one link from each leg;
- Defining weights for the different body parts consistently with the real ANYmal robot;
- Changing torso shape to a rectangle;

and are summarised in section 4.1. In addition to the changes to the simulation, I also implemented better PD control (section 4.2), improved my methods for evaluating reward functions and improved the reward function itself (section 4.3).

4.1 HalfANYmal model

Table 4.1 shows the dimensions of the HalfANYmal model that I implemented using MuJoCo's MJCF XML format. The values are based on the information kindly provided by Dr. Zhibin Li, my supervisor. The only significant change I made was to the torso mass. This was originally 21.39 kg (plus 1.1 kg per hip motor) but that is the mass of ANYmal's torso. Since HalfANYmal only has half the number of legs and half the number of motors (and consequently half the strength) compared to ANYmal, the torso mass also needed to be halved. Otherwise, HalfANYmal would have trouble carrying its weight and each of its feet would experience higher friction with the floor. Hence, the torso mass of HalfANYmal is $21.39/2 + 2 \times 1.1 = 12.895$ kg.

At first, I received some contradictory pieces of information regarding the maximum motor torque. I settled for 40 Nm maximum for each motor (both knee and hip) which is what Hutter et al. [2016] report. To be precise, this is the maximum torque that ANYmal can deliver continuously but it can spike higher momentarily. The HalfANYmal

Figure 4.1: The ANYmal robot. (Picture retrieved from `https://www.anybotics.com/wp-content/uploads/2018/04/ANYmal-Bedi-1030x999.png`.)

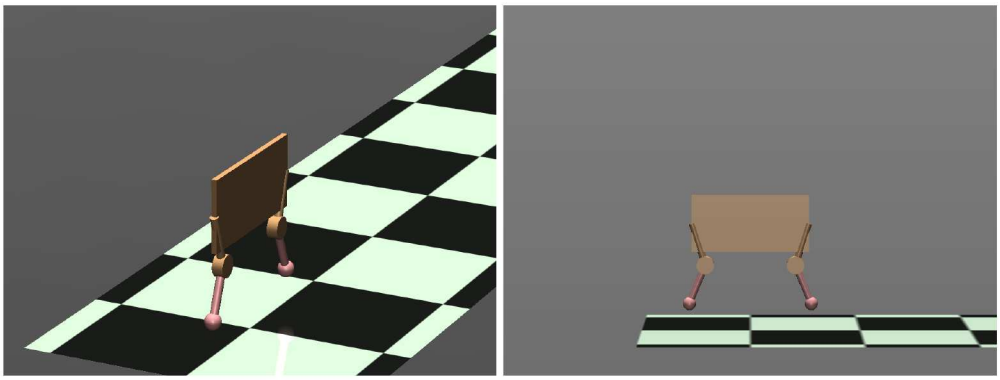

Figure 4.2: The HalfANYmal environment

	Shape & Dimension	Count	Mass [kg]	
			Unit	Total
Torso	Box of size 0.531 m by 0.26 m	1	12.895	12.895
Thigh	Cylinder of 0.2 m length and 0.015 m radius	2	0.318	0.636
Knee motor	Cyclinder of length 0.08m and radius 0.04 m	2	1.1	2.2
Shank	Cylinder of 0.19 m length and 0.015 m radius	2	0.318	0.636
Foot	Sphere of radius 0.03 m	2	0.19	0.38
TOTAL		9		16.747

Table 4.1: Dimensions of HalfANYmal body parts

does not capture this since I did not know how to implement it in MuJoCo. Similarly, my model does not realistically reflect the time that ANYmal needs to change the output torque. This could be improved in the future.

ANYmal's joints are capable of rotating through a full revolution and are only limited by the cabling that crosses them. Hence, I set the joint limits for HalfANYmal also just above a full revolution. Self-collisions are not enabled in my simulation. The real ANYmal can tilt its legs slightly to keep them from colliding but since HalfANYmal cannot do that, I disabled the collision detection.

The HalfANYmal environment operates at 20 Hz (i.e. each time step corresponds to 50 ms[1]) and the underlying physics simulation runs at 200 Hz. The environment does not have any terminal states but uses fixed-length episodes of $1,000$ time steps.

4.2 Improved PD control

I have also found a better way to implement PD control for the HalfANYmal environment. MuJoCo computes gravitational forces on all the joints which can be obtained easily and used for gravity compensation. I also took joint positions and velocities from the simulator and used them to implement PD in python as

$$\tau(\mathbf{x}) = K_p(\mathbf{x} - \alpha) - K_d\dot{\alpha} + \mathbf{g} \tag{4.1}$$

where K_p and K_d are the PD gains, $\mathbf{x}$ are the (joint position) inputs, α are the current joint positions, $\dot{\alpha}$ are the joint velocities and $\mathbf{g}$ contains the gravity biases.The $\mathbf{x}$ is supplied by the agent while α, $\dot{\alpha}$ and $\mathbf{g}$ are supplied by the MuJoCo simulator. The torques $\tau(\mathbf{x})$ are sent as inputs for the MuJoCo model. The values of the PD gains are summarised in table 4.2).

I repeated the same experiment as for the HalfCheetah model and used it to manually tune the PD gains. The resulting performance is shown in figure 4.3. Same as the one from section 3.3, this implementation of PD control eliminates virtually all oscillations and overshoot. However, thanks to gravity compensation, it achieves even slightly

[1]This is not necessarily real-world wall clock time since the simulation can run slower or faster. During training, the simulator usually "speeds up the flow of time" by up to several orders of magnitude.

Joint	back hip	back knee	front hip	front knee
K_p	100	140	100	140
K_d	8	10	8	10

Table 4.2: PD gains for HalfANYmal

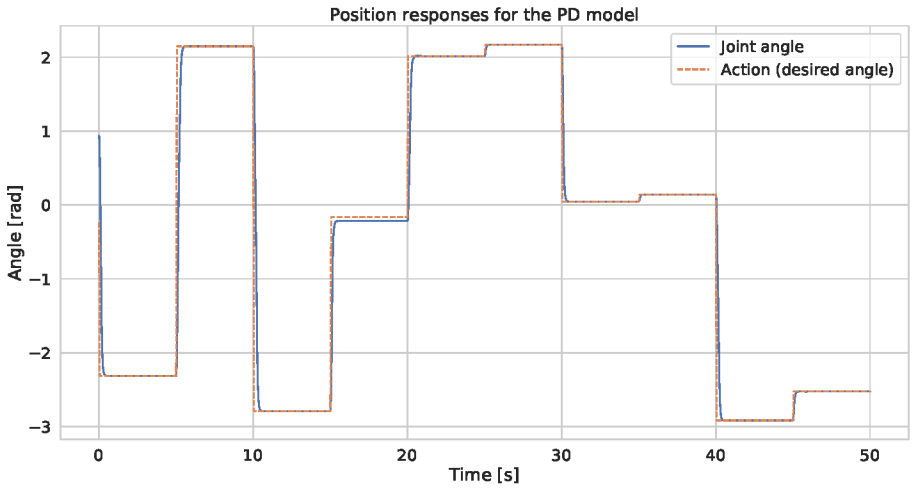

Figure 4.3: Position responses for the back knee joint of HalfANYmal.

lower steady-state error. But most importantly, it does this with much lower PD gains and without relying on unrealisticly high torques.

With this setup, I achieved balanced locomotion for the HalfANYmal robot as can be seen in this video[2]. However, the agent learned a very strange gait where it essentially uses its entire shanks as feet, with the knees playing the role of heels. Hence both the knees and the torso frequently impact the ground. While ANYmal should be sturdy enough to survive such gait, it is not meant to be used like that. I will try to improve the behaviour in the next chapter, but it is possible that HalfANYmal is not capable of a "nicer" gait since, unlike ANYmal, it cannot lift one of its legs without falling.

4.3 Reward shaping

In section 3.1, I improved the interpretability of rewards by restricting them to the range $(0, 1]$ using RBFs. This approach is also extensible, allowing more metrics to be taken into account. For example, a reward for keeping the torso at certain height above the ground can be introduced to discourage the gait described at the end of last section. In section 3.1, I was evaluating different reward functions only by visually inspecting the the video of a trained agent which is a bad method for several reasons:

- It is inherently subjective and it becomes very difficult with growing number of

[2] https://youtu.be/xFkq8YatovE

experiments;

- It cannot take into account small and subtle changes;
- It cannot distinguish between small changes caused by the different reward function and those caused by different random initialisation of an episode;
- It cannot see the effects of different reward functions on the learning process.

To address these issues, I developed code to record all the joint velocities and positions, and all the motor torques during an episode, as well as code to record the individual reward components (e.g. reward for velocity or orientation) throughout training. The former allows me to take two trained agents, run them for one or more episodes and precisely compare things like what velocity do they achieve or how much torque they use. The latter allows me to check if the learning algorithm is actually picking up on all the reward components and how "hard to learn" those components are (i.e. how long does the algorithm take to figure out how to collect them).

Unfortunately, both of these improvements were tricky to implement due to the poor structure of the OpenAI Baselines library. For example, I had to change small bits of code in many different places in order to log the few extra pieces of information I was after during the training. The library also did not allow me to execute two different trained agents in a single Python process (this probably has something to do with TensorFlow's static graph model) so I had to use multiprocessing to execute every agent. However, this at least allowed me to run the agents in parallel which sped up the comparisons.

With everything implemented, I set out to do three things:

1. Find a velocity that is easy to reach and maintain as well as the maximum velocity for the HalfANYmal robot;
2. Check my hypothesis from section 3.1 about the narrow torque reward being difficult to pick up;
3. Try to prevent HalfANYmal from walking on its knees by introducing reward for height above ground.

4.3.1 Velocity experiments

I made another small tweak to the velocity component of the reward function, which is now defined as

$$R = 0.7e^{-\left(\frac{v}{v_{desired}}-1\right)^2} + 0.2e^{-\alpha^2} + 0.1e^{-||\mathbf{t}||^2}. \tag{4.2}$$

This helps with interpretability of the results, especially the intermediate ones during training, since the initial reward for not moving is independent of the desired velocity. It also makes sure that the agent never starts far out on flat tail of the RBF where it would not detect the reward signal. I used the default torque term here since I will be trying to find the optimal width of the torque RBF in the next section.

To investigate what velocities the HalfANYmal robot is capable of, I trained one agent for each value of $v_{desired} \in \{0.5, 0.8, 1.0, 1.5, 2.0, 3.0\}$ for 20,000,000 time steps. During training, I recorded the evolution of all the individual reward components as described previously. After the training, I ran each of the trained agents for 10 episodes, recording the forward velocity (and also all other velocities, positions and torques) and averaging it over the 10 episodes.

Figure 4.4 shows the forward velocities achieved by the different agents. The only agent that actually achieved its target velocity was the one with $v_{desired} = 0.5$. The agent trained with $v_{desired} = 1$ also achieved stable locomotion but only at around 0.7 m/s. Only slightly higher velocity was achieved by the agents with $v_{desired} = 1.5$ and $v_{desired} = 3$. Interestingly, the agent for $v_{desired} = 2$ achieved significantly higher velocity of over 1.2 m/s.

Also interesting is the poor performance of the agent with $v_{desired} = 0.8$ which was slower than 0.5 m/s. Perhaps, there is a border between two different gaits which the $v_{desired} = 0.8$ agent did not manage to cross but the $v_{desired} = 1$ one did, but it is also possible that $v_{desired} = 0.8$ agent just found an unlucky local optimum. Inspecting the videos suggests the latter. The $v_{desired} = 0.5$ agent[3] shows a sort of crawling motion which can probably be made arbitrarily slow but not arbitrarily fast. Indeed, the $v_{desired} = 1$ agent[4] shows similar kind of motion, but faster, occasionally even leaving the ground which would normally be a sign of running. The $v_{desired} = 0.8$ agent[5] tries to perform a similar gait but sometimes moves its rear leg too much and gets stuck for a while. The quickest agent ($v_{desired} = 2$) shows a distinctly different gait[6] which suggests that the $v_{desired} = 1$ agent has probably hit the limit of the crawling gait and that is why it did not move faster.

The velocity learning curves in figure 4.5 show that it becomes more difficult to find an optimal policy when $v_{desired} > 1$. This makes sense since, according to Hutter et al. [2016], the ANYmal robot can achieve maximum speeds of around 1 m/s.

In conclusion, $v_{desired} = 1$ and $v_{desired} = 2$ seem to be the best settings for achieving high speed. Increasing $v_{desired}$ more actually results in significantly lower speed so it seems like the maximum speed that the HalfANYmal robot is capable of is about 1.4 m/s.

4.3.2 Torque experiments

Figure 4.6 shows that, in the experiments from the previous section, the torque reward component was always basically 0 so the learning algorithm did not have a chance to pick up on it which confirms my hypothesis from section 3.1.

To help the agent find the reward, I tried widening the torque RBF. I used the reward

[3] https://youtu.be/WFWIhLSBewo
[4] https://youtu.be/h9WL2HV7MnM
[5] https://youtu.be/FEC74oopvdE
[6] https://youtu.be/XBNsz4PWqN8

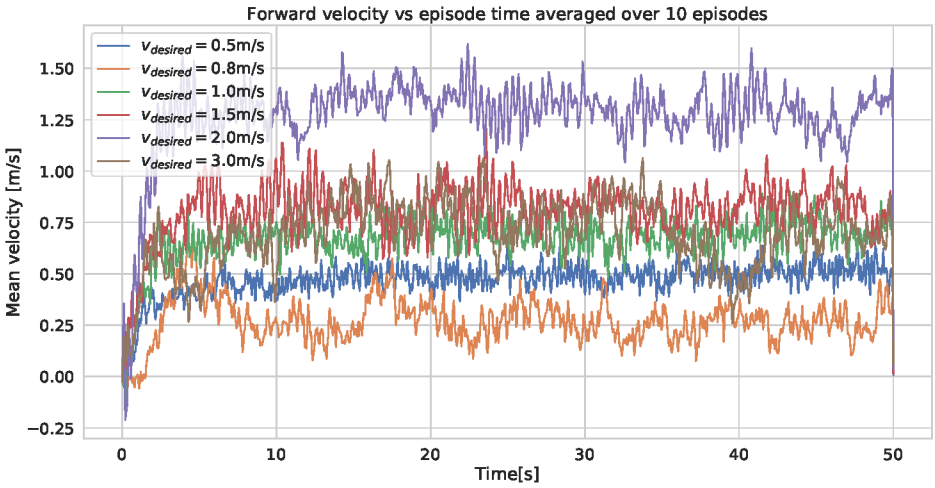

Figure 4.4: Forward velocities during an episode averaged over 10 episodes for agents trained for different values of $v_{desired}$.

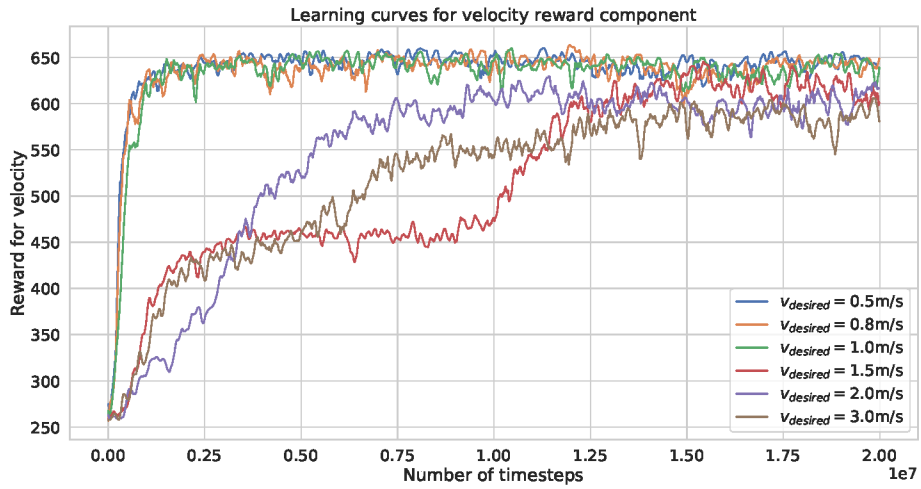

Figure 4.5: Velocity learning curves for agents trained for different values of $v_{desired}$. Each point on the plot shows reward accumulated over an entire episode of 1000 time steps. The maximum value is 700 since the weight of the velocity component of the reward function (4.2) is 0.7.

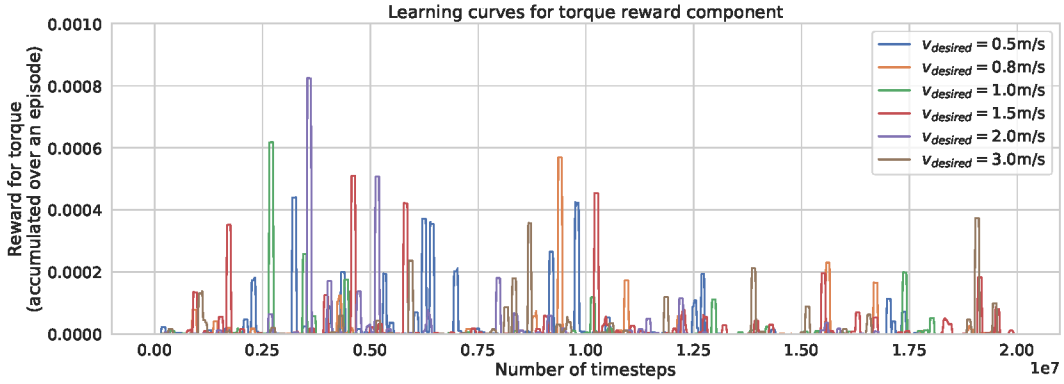

Figure 4.6: Torque learning curves for agents trained for different values of $v_{desired}$ in section 4.3.1. The maximum reward is 100 since the weight of the torque component of the reward function (4.2) is 0.1.

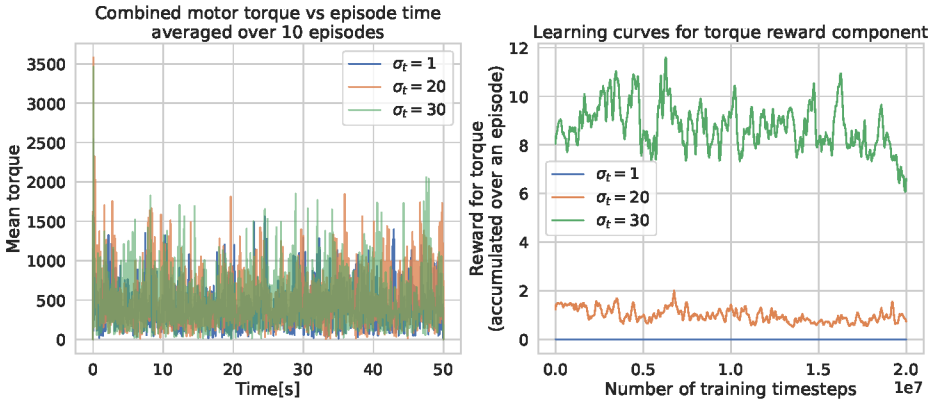

Figure 4.7: Motor torques and torque learning curves for agents trained for different values of σ_t. The plot on the left shows squared magnitude of the vector of all motor torques. The maximum value of the torque reward is 100 since the weight of the torque component of the reward function (4.3) is 0.1.

function

$$R = 0.7e^{-\left(\frac{v}{v_{desired}}-1\right)^2} + 0.2e^{-\alpha^2} + 0.1e^{-\left(\frac{||t||}{\sigma_t}\right)^2}. \tag{4.3}$$

where σ_t corresponds to the width of the RBF and can be thought of as a "tolerance" parameter. I kept $v_{desired} = 1$ since that showed good performance in the previous section and I tried the values of $\sigma_t \in \{1, 20, 30\}$ (note that $\sigma_t = 1$ has already been tried in the previous section so I only need to perform two new experiments).

Figure 4.7 shows that none of the experiments was successful in helping the agent pick up the reward. There was no significant change in the amount of torque used by the trained agent during an episode and the learning curves show that the learning algorithm did not try to optimise for torque. However, reward received was higher and it changed noticeably with changing policy so the learning algorithm should have been able to detect it (especially for $\sigma_t = 30$). It shows that the weight of the torque term is likely too low and so the algorithm simply ignored it and optimised for velocity (as there is an inherent trade-off between the two).

4.3.3 Height experiments

In an attempt to prevent HalfANYmal from walking on its knees, I tried to add a reward component that would encourage the agent to keep its torso higher above the ground:

$$R = 0.7e^{-\left(\frac{v}{v_{desired}}-1\right)^2} + 0.1e^{-\alpha^2} + 0.1e^{-||\mathbf{t}||^2} + 0.2e^{-\left(\frac{h-h_{desired}}{0.05}\right)^2} \tag{4.4}$$

[7]where h is the height of the centre of the torso above the ground. The reward function does not include the updated torque term as I performed the two sets of experiments in parallel.

To choose $v_{desired}$, I looked at the heights achieved in the experiments of section 4.3.1. Figure 4.8 shows that the agent trained with $v_{desired} = 3$ achieved the highest height above the ground. Additionally, $v_{desired} = 1$ has achieved stable results in terms of velocity. Hence I will use $v_{desired} = \in \{1,3\}$ in this set of experiments.

If HalfANYmal stood still with both its legs straightened and pointed down, the centre of its torso would be $h = 0.42$ m above the ground. On the other hand, if the torso is level and $h > 0.22$ m, the knees cannot touch the ground. Hence, in this set of experiments, I will use $h_{desired} \in \{0.3, 0.35, 0.4\}$ for each setting of $v_{desired}$.

This set of experiments yielded some surprising results. The height learning curves in figure 4.9 looked promising at first since all the configurations except $h_{desired} = 0.4, v_{desired} = 1$ achieved close to the highest possible reward. However, the actual heights shown in figure 4.10 contradict this since two different agents maintained the height of around 0.13 m which corresponds exactly to HalfANYmal lying on its belly (or back) and gives reward very close to 0.

It was very difficult to explain this conflict. Eventually, I found out that the episode reward values reported by OpenAI Baselines were actually smoothed and even towards the end of the training, there were still episodes in which the performance dropped back to zero as shown in the background of figure 4.9. It is possible that OpenAI Baselines does not save the best policy but rather the last one (although this is rather difficult

[7]An attentive reader may notice that the weights of the RBFs add up to 1.1 rather than 1. This was not intentional and I only noticed it after running all relevant experiments. Fortunately, this only matters for the interpretability of the total reward. The training process only depends on the relative sizes of the weights and any small change to this is as good a guess as any other. On the other hand, this mistake does make it slightly easier to compare the velocity reward with the previous experiments. I planned to use 0.6 instead of 0.7 for the velocity weight.

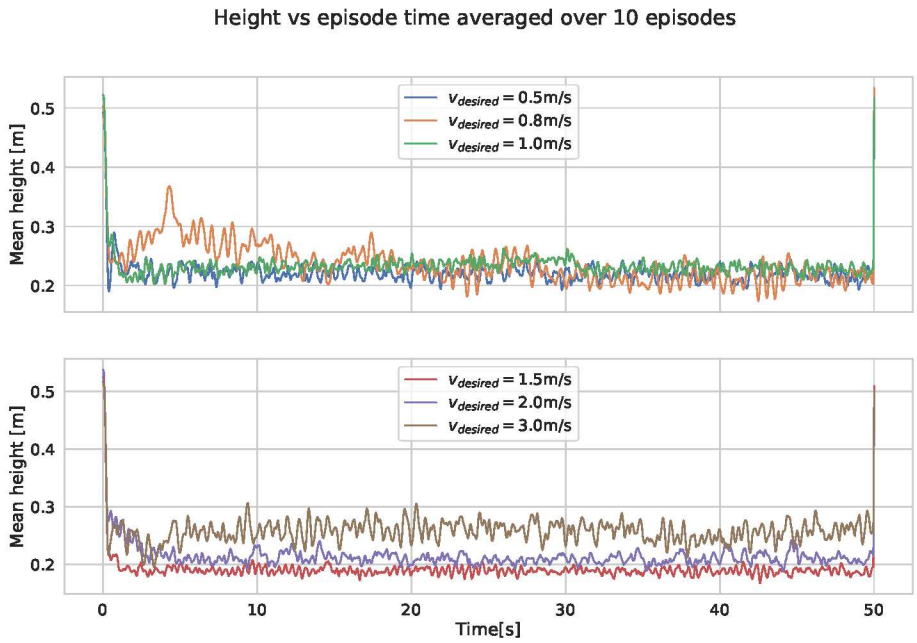

Figure 4.8: Heights above ground for agents trained for different values of $v_{desired}$ in section 4.3.1.

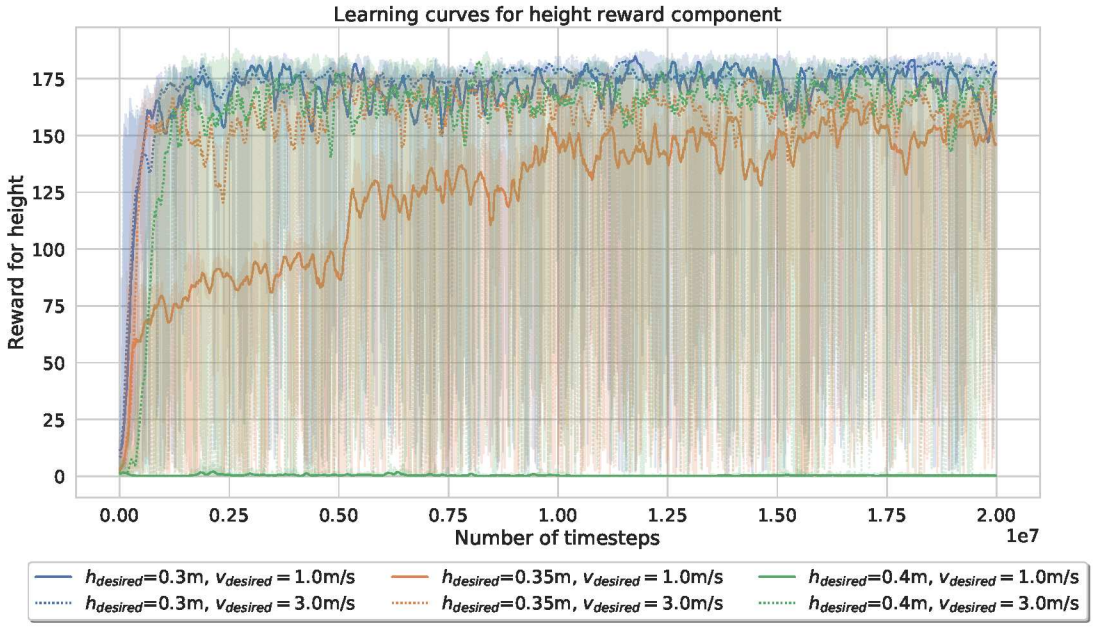

Figure 4.9: Height learning curves for agents trained for different combinations of $v_{desired}$ and $h_{desired}$. Raw per-episode reward before smoothing is shown in the background. The maximum value of the height reward is 200 since the weight of the height component of the reward function (4.4) is 0.2.

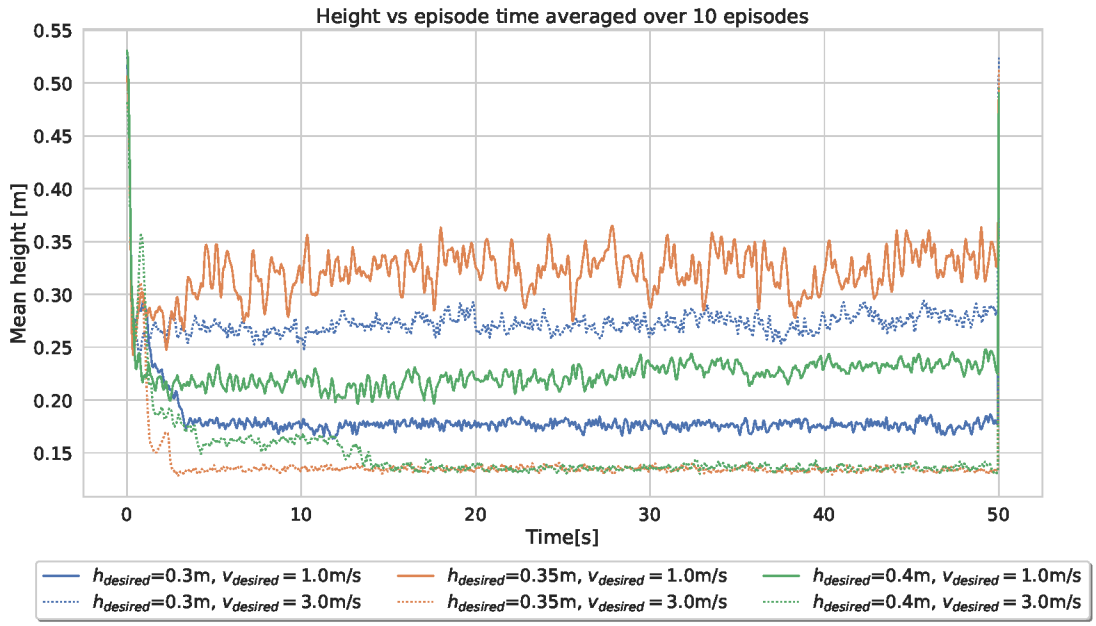

Figure 4.10: Heights above ground for agents trained for different combinations of $v_{desired}$ and $h_{desired}$.

to know for sure as it is not documented anywhere and the relevant code is extremely convoluted), which would explain the aforementioned conflict - some agents simply got unlucky with a bad last update to their policies.

However, there was one positive result. As shown in figure 4.10, the $h_{desired} = 0.35, v_{desired} = 1$ configuration managed to maintain height between 0.3 and 0.35 for most of the time. While the overall performance of the agent was unstable, it sometimes managed to make a few steps forward without touching the ground with its knees or torso as can be seen in this video[8]. As can be seen from the learning curves in figure 4.9, this configuration learned significantly slower than others and may not have had enough time to fully converge. It is therefore possible that stable walking could be achieved with longer training. Nonetheless, this result proves that HalfANYmal is physically capable of walking without hitting the ground with its knees or torso.

[8] https://youtu.be/dS2K4_84BlM?t=8

Chapter 5

Learning Dynamic Locomotion on Rugged Terrain

The last goal of the first part of this MInf project was to train HalfANYmal to walk or run on uneven terrain. I modified and tested the HalfANYmal environment to include such terrain and then, building on the reward shaping experience from the previous chapter, I trained a few agents for the modified environment.

5.1 Environment modifications

In this section I describe how I generated the uneven terrain, modified the simulation to include it and also modified the Python environment to enable the agent to sense the terrain around itself.

To implement the uneven terrain in the simulation, I used MuJoCo's height field. This allows me to specify an array of heights and the simulator will create the corresponding surface, interpolating linearly between the specified points. I used 10 cm resolution. Since HalfANYmal's track is 100 m long, it essentially consists of $1,000$ rectangles. Each rectangle is either level or at a slope. For any two adjacent rectangles, at least one of them is level. The height difference between the two ends of a sloped rectangle is approximately between 5 and 12 cm.

Normally, a robot would need some sensor(s) to detect the terrain around it and react to it. Since I already have the ground truth, in each time step, I simply sample the heights of a 5 m long segment of the track starting about 0.7 m behind the current position of the HalfANYmal at 5 cm intervals. For example, a lidar sensor should be able to provide similar data (I think), with the major difference being that my method does not take into account potential occlusions.

Ideally, a new random height profile would be generated for every episode so that the agent cannot just memorise the track but needs to learn to actually react to the different

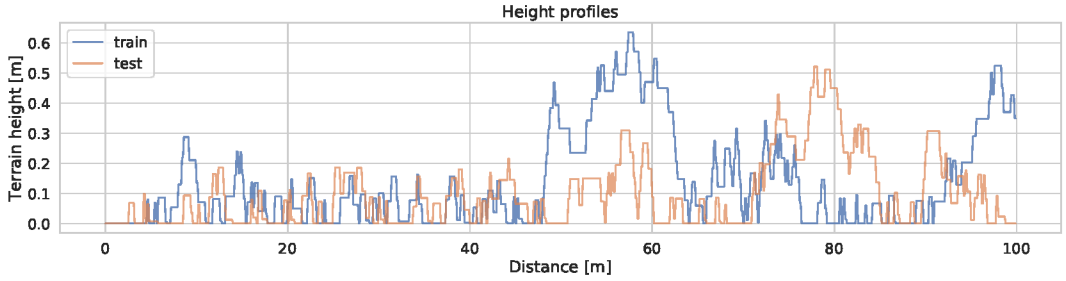

Figure 5.1: Training and testing terrain profiles.

terrains. However, I was not able to implement this in the Python wrapper for MuJoCo[1] that OpenAI Gym uses so I only generated the terrain once and used it for all training. But since the agent only sees a small portion of the track at any given time and cannot observe the distance along the track that it has travelled, it is still possible that it will learn to generalise. Seeing if it does is in itself an interesting experiment as it can save a small amount of computation and complexity if the terrain does not need to be generated after every episode. For this experiment, I generated another terrain to test the trained agents on something that they have not seen during training. The height profiles of both terrains can be seen in figure 5.1.

5.2 Baseline experiments

Since I did not know, what performance to expect from the agents on the new uneven terrain, I needed to establish a baseline. To do that, I ran the best agents from chapter 4 (trained on flat ground) on the training terrain[2]. In theory, agents trained on the uneven terrain should be able to achieve much higher performance than those trained on flat ground. I chose to compare the agents based on distance travelled during an episode since velocity plots would be very noisy due to the terrain and velocity can still be seen in the distance plots as the slope of the curve.

I ran each agent for 20 episodes and averaged the results for better reproducibility. Figure 5.2 shows that the agent trained with $v_{desired} = 1$ and $\sigma_t = 30$ in section 4.3.2 managed to travel the longest distance of just over 20 m (see video[3]). On flat ground, this agent achieved speeds of about 0.7 m/s which would have it travelled around 35 m. Hence the uneven terrain slowed it down but not very much, which is quite impressive. Note that the fastest agent from chapter 4 ($v_{desired} = 2$ which reached over 1.2 m/s in section 4.3.1) would be able to travel around 60 m on the flat ground, but, on average, only travelled 10 m on the uneven terrain.

Perhaps, the good performance should not be too surprising here. After all, legged

[1] MuJoCo is a C++ framework.

[2] This requires tiny modification to the new environment to disable the sensing of the terrain. That is, the state of the environment (called observation in the code) must not include the terrain heights. Otherwise, the observations have wrong dimension and the agents cannot process them.

[3] `https://youtu.be/VeRvv5Gl3PQ`

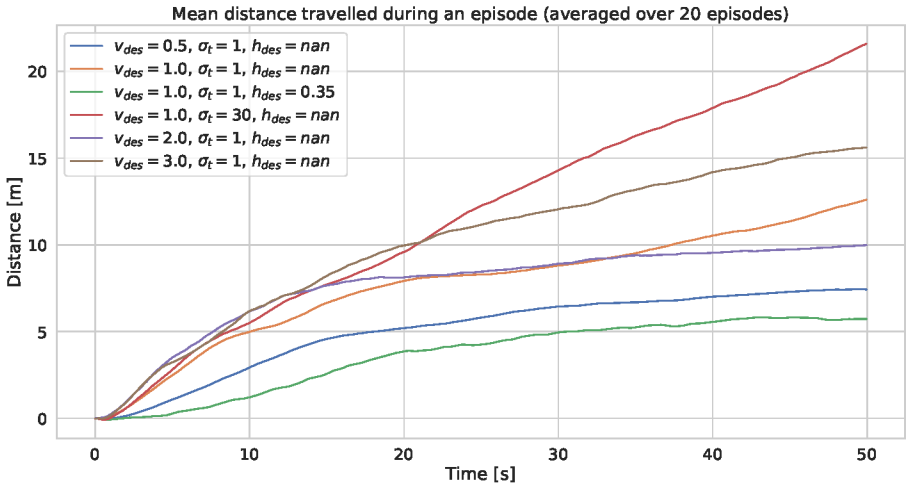

Figure 5.2: Performance of some of the agents trained on flat ground in chapter 4 and evaluated on the uneven training terrain from this chapter.

things like humans and animals are much better at getting through difficult terrains like mountains or other obstacles than wheeled or tracked vehicle. The RHex robot from Boston Dynamics also demonstrates[4] that clever limb design can be very effective at negotiating difficult obstacles even without an advanced control algorithm.

5.3 Results

In this section I present the results of training 8 different agents on the training terrain shown in figure 5.1. I trained each agent for $30,000,000$ time steps since navigating rugged terrain is likely a harder task than walking or running on a flat surface. For the first 6 agents, I used the 6 reward functions of the baseline agents described in the previous section.

Up until now, all the reward functions were primarily based on velocity which is fine for flat surface but different velocities might be appropriate for different terrains (e.g. steep/shallow downhill/uphill). Hence, I trained the last 2 agents with a reward function based on distance travelled as

$$R = 0.9e^{-\left(\frac{d}{d_{target}}-1\right)^2} + 0.02e^{-\left(\frac{v}{1}-1\right)^2} + 0.03e^{-\alpha^2} + 0.02e^{-\left(\frac{||\mathbf{t}||}{30}\right)^2} + 0.03e^{-\left(\frac{h-0.35}{0.05}\right)^2} \tag{5.1}$$

where d is the distance travelled and $d_{target} \in \{50, 80\}$ is the target distance. It is possible that the agent would stop before reaching d_{target} since the wide RBF becomes very flat around there. The 50 m target should be enough to clearly beat the 20 m baseline and I added the 80 m target to see if the agent can match the performance achieved on flat surface (unlikely).

[4] https://youtu.be/ISznqY3kESI

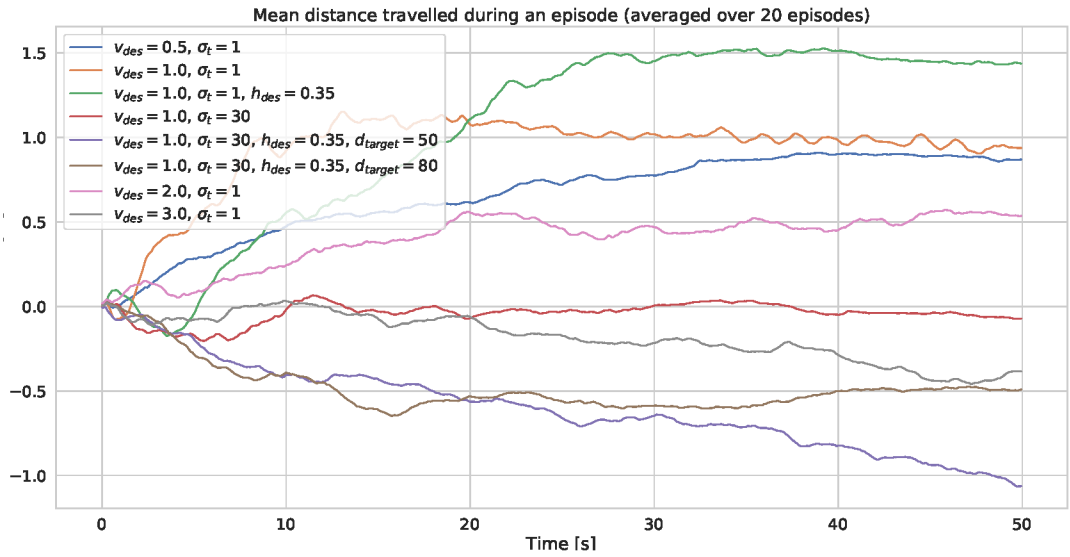

Figure 5.3: Performance of the agents trained and evaluated on the uneven (training) terrain.

I put a very high weight (0.9) on the distance term because the optimal values of all the other quantities will depend on the local terrain so I do not want to constrain the agent too much. The other terms are mostly there to help the agent get started since the distance reward is so flat that the agent might need to travel several meters to realise that moving forward is desirable. I chose the desired velocity and height based on what worked well in the previous chapter.

Interestingly, none of the agents trained on the uneven terrain managed to beat or even match the baseline despite training for longer. Figure 5.3 shows that the best agent only managed to travel 1.5 m on average and many agents even moved backwards. The learning curves in figure 5.4 show that the agents were not able to learn anything useful. Three agents picked up a little bit of the velocity reward but this was not enough to travel anywhere far. One agent picked up much of the reward for height which is consistent with section 4.3.3. None of the agents was able to optimise for torque, consistently with section 4.3.2 and all agents seem to have picked up the reward for orientation which is also consistent with the previous results (section 4.3). However, the two agents with distance-based reward functions showed significantly noisier orientation learning curves which might be a sign of them not really optimising for orientation and simply ending up in the correct orientation by random chance since it is the initial and most probable orientation. This would also suggest that the weight coefficients of the reward function 5.1 may have been too aggressive. However, despite the high weight on it, the agent did not pick up on the distance reward.

These results suggest that the task of learning to negotiate uneven terrain is too difficult for the algorithm used. It is possible that different hyperparameter settings could achieve better results. Another way to improve the performance could be with transfer learning as defined by Tan and Cheng [2009]. This means to use the neural network weights learned by an agent on a flat surface to initialise the networks of a new agent before training it on the uneven terrain. This way, the new agent would be able to crawl

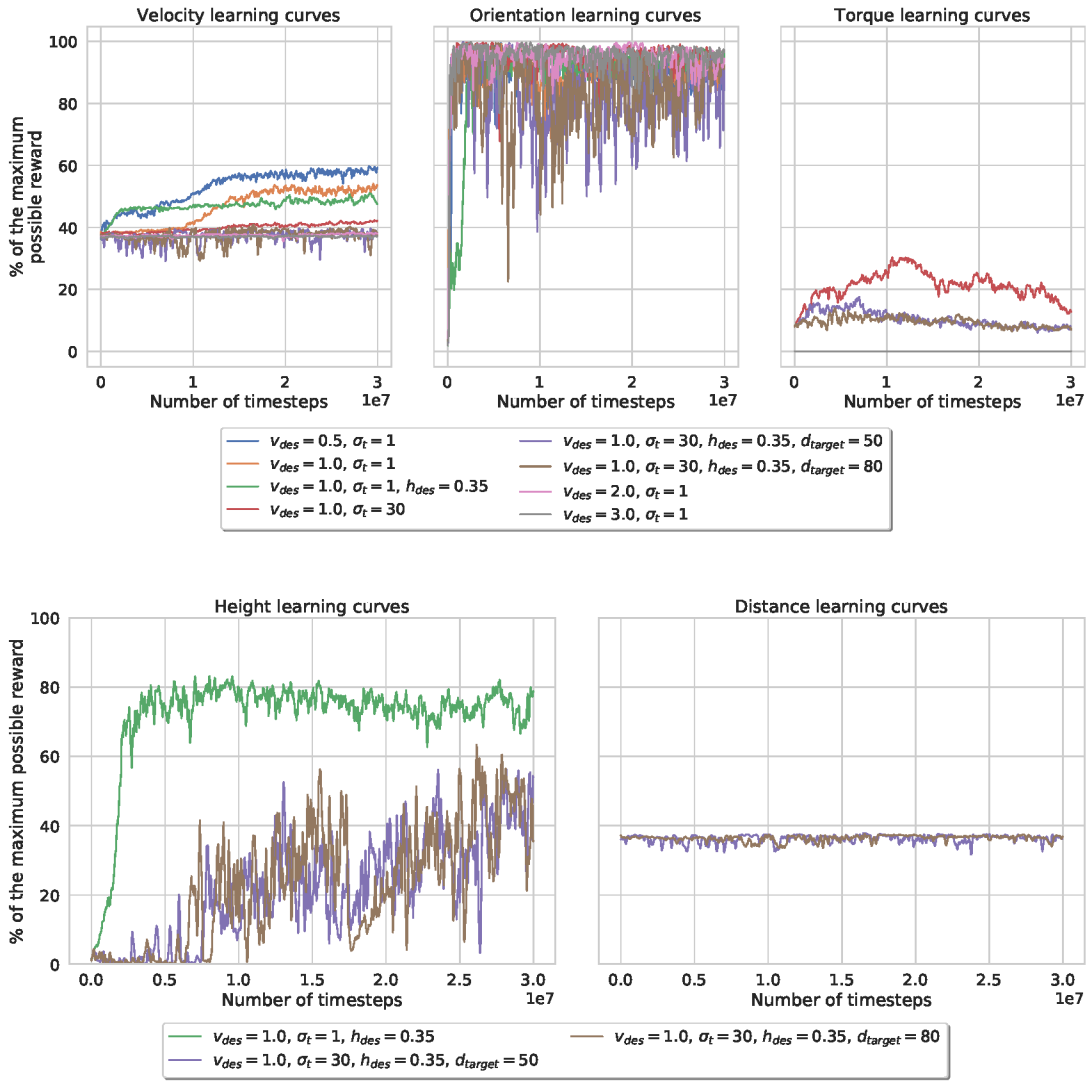

Figure 5.4: Learning curves for the agents trained and evaluated on the uneven (training) terrain.

over some obstacles to begin with which should enable it to gather more experience and find more efficient ways of negotiating obstacles. This should only require a small change to the code to account for the different dimensionalities of the state spaces but it would be very difficult to do with the OpenAI Baselines implementation. Nonetheless, the baseline results still demonstrate that HalfANYmal is capable of negotiating uneven terrain.

Chapter 6

Conclusions and Future Work

In this first part of the MInf Project, I achieved all the goals I set out to achieve. As described in section 4.1, I modified the HalfCheetah environment from OpenAI Gym to resemble the ANYmal robot, creating the HalfANYmal environment. In section 4.3, I successfully used reinforcement learning to teach HalfANYmal to walk forwards. Although the resulting motion was very clumsy, I showed that HalfANYmal is physically capable of a "nice" walking gait and hence more research into achieving such gait with reinforcement learning would be warranted.

Finally, in chapter 5, I showed that a HalfANYmal robot trained with reinforcement learning is capable of negotiating uneven terrains. The relevant experiments yielded some very unexpected results where agents trained on flat ground performed quite well on uneven terrain yet agents trained on uneven terrain failed to learn any useful movements. This might suggest that learning to walk at all is actually significantly more difficult, than walking over obstacles once an agent knows how to walk, but uneven terrain makes learning to walk at all much harder.

6.1 Future work (for MInf Project Part 2)

In Part 1 of this MInf project, I focused on developing the simulation side of things and produced a functional and tested HalfCheetah environment. In Part 2, I would like to focus more on the machine learning and RL aspect and find methods to train better performing agents faster and more efficiently.

The surprising results of chapter 5 provide a great ground for future research. As suggested at the end of that chapter, I would like to investigate the use of transfer learning in RL to leverage the knowledge learned by agents on a flat ground to train agents on uneven terrains. Taylor and Stone [2009] show that using transfer learning for RL is indeed a viable idea. Specifically, I will seek answers to the following questions:

- Does training an agent on uneven terrain with knowledge transferred from a flat-surface agent improve, or at least retain the performance of the flat-surface agent on the uneven terrain?

- Can transfer learning shorten training time or improve sample efficiency?
- Are there other places where knowledge transfer could help? For example, does learning how to stand still with torso at a certain height help with learning how to walk while keeping the torso at that height?

Smaller improvements may include making sure to save the best model found during training rather than the last one (as discussed in section 4.3.3) and training the promising agent from section 4.3.3 for longer. I can also make a more thorough comparison of the PPO and DDPG algorithms discussed in section 2.1.1 and potentially use DDPG if it achieves significantly better performance.

Further research could also be done into evaluation of performance of different agents and comparing different reward functions. As discussed in chapter 4, watching the videos of trained agents is not scalable and I developed other methods, namely monitoring different components of the reward during training and plotting the simulation ground truth data of quantities of interest during evaluation. However, I was not able to find any formalised and tested methods in the current literature.

Bibliography

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

DeepMind Technologies. The deepmind control suite and package, 2019. URL https://github.com/deepmind/dm_control.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016. URL http://arxiv.org/abs/1604.06778.

Marco Hutter, Christian Gehring, Dominic Jud, Andreas Lauber, C. Dario Bellicoso, Vassilios Tsounis, Jemin Hwangbo, Karen Bodie, Peter Fankhauser, Michael Bloesch, Remo Diethelm, Samuel Bachmann, Amir Melzer, and Mark A. Höpflinger. Anymal - a highly mobile and dynamic quadrupedal robot. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016*, pages 38–44, 2016. doi: 10.1109/IROS.2016.7758092. URL https://doi.org/10.3929/ethz-a-010686165.

Zhibin Li, Chengxu Zhou, Qiuguo Zhu, and Rong Xiong. Humanoid balancing behavior featured by underactuated foot motion. *IEEE Transactions on Robotics*, 33(2): 298 – 312, 4 2017. ISSN 1552-3098. doi: 10.1109/TRO.2016.2629489.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016. URL http://arxiv.org/abs/1509.02971.

OpenAI. Openai gym, 2018. URL https://github.com/openai/gym.

Marc Raibert, Kevin Blankespoor, Gabriel M. Nelson, and R. Playter. Bigdog , the rough-terrain quadruped robot. 2008. URL https://pdfs.semanticscholar.org/86cf/98c8ad25b3cb185b7524bccf32f77a91b616.pdf.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Alexander Spröwitz, Alexandre Tuleu, Massimo Vespignani, Mostafa Ajallooeian, Emilie Badri, and Auke Jan Ijspeert. Towards dynamic trot gait locomotion: Design, control, and experiments with cheetah-cub, a compliant quadruped robot. *I. J. Robotics Res.*, 32:932–950, 2013.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive computation and machine learning series. MIT Press, Cambridge, MA, 2018. http://incompleteideas.net/book/the-book.html.

Songbo Tan and Xueqi Cheng. Improving scl model for sentiment-transfer learning. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers*, NAACL-Short '09, pages 181–184, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1620853.1620903.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller. Deepmind control suite. *CoRR*, abs/1801.00690, 2018. URL http://arxiv.org/abs/1801.00690.

Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685, 2009.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109. URL https://doi.org/10.1109/IROS.2012.6386109.

Pengfei Wang, Yapeng Shi, Fusheng Zha, Zhenyu Jiang, Xin Wang, and Zhibin Li. An analytic solution for the force distribution based on cartesian compliance models. *International Journal of Advanced Robotic Systems*, 16(1), 2019. ISSN 1729-8806. doi: 10.1177/1729881419827473.

Chuanyu Yang, Taku Komura, and Zhibin Li. Emergence of human-comparable balancing behaviors by deep reinforcement learning. *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 372–377, 2017. URL http://arxiv.org/abs/1809.02074.

Chuanyu Yang, Kai Yuan, Wolfgang Merkt, Taku Komura, Sethu Vijayakumar, and Zhibin Li. Learning whole-body motor skills for humanoids. *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, pages 270–276, 2018. URL https://homepages.inf.ed.ac.uk/svijayak/publications/yang-Humanoids2018.pdf.

Kai Yuan, Iordanis Chatzinikolaidis, and Zhibin Li. Bayesian optimization for whole-body control of high degrees of freedom robots through reduction of dimensionality. *IEEE Robotics and Automation Letters*, 2 2019. ISSN 2377-3766. doi: 10.1109/LRA.2019.2901308.