UNIVERSITY OF ŽILINA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

MASTER THESIS

BC., DANIEL ADAMKOVIČ

Realizing walking for a walking robot using deep reinforcement learning

Supervisor: doc. Ing. Michal Gregor, PhD.

Document ID: 28260220212001

Žilina, 2021

UNIVERSITY OF ŽILINA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

MASTER THESIS

BC., DANIEL ADAMKOVIČ

Realizing walking for a walking robot using deep reinforcement learning

Study programe: Cybernetics

Field of study: 5.2.14 Automation

Supervising institution: University of Žilina,
Faculty of Electrical Engineering and Information Technology,
Department of Control and Information Systems

Supervisor: doc. Ing. Michal Gregor, PhD.

Žilina, 2021

ŽILINSKÁ UNIVERZITA V ŽILINE
Fakulta elektrotechniky a informačných technológií
Katedra riadiacich a informačných systémov

Akademický rok 2020/2021 Evidenčné číslo **28260220212001**

ZADANIE DIPLOMOVEJ PRÁCE

Meno študenta: **Bc. Daniel Adamkovič**

Študijný program: **Riadenie procesov**

Téma diplomovej práce: **Realizácia kráčania pre kráčajúceho mobilného robota pomocou hlbokého učenia s odmenou**

Pokyny pre vypracovanie diplomovej práce:

1. Vypracovanie prehľadu o metódach hlbokého učenia a metódach hlbokého učenia s odmenou (DRL), s dôrazom na metódy umožňujúce spojité riadenie (policy-based metódy).
2. Výber vhodného simulačného nástroja, implementácia modelu robot, tréningovej úlohy a odmeňovacej funkcie.
3. Výber existujúcej implementácie zvolenej DRL metódy (v prípade potreby jej úprava alebo tvorba vlastnej implementácie).
4. Tréning DRL agenta na realizáciu kráčania pre kráčajúceho mobilného robota v simulovanom prostredí.
5. Verifikácia a zhodnotenie riešenia; návrh odporúčaní na transfer riešenia zo simulácie do reálneho sveta.

Vedúci diplomovej práce : **doc. Ing. Michal Gregor, PhD.**

Dátum odovzdania diplomovej práce : **3. 5. 2021**

V Žiline dňa 4. 12. 2020

[illegible]

prof. Ing. Juraj Spalek, PhD.
vedúci katedry

Acknowledgments

I would like to thank my supervisor doc. Ing. PhD. Michal Gregor, for his help in choosing the topic of this thesis and for his advice and constructive criticism, without which the quality of this work would have surely suffered. I would also like to thank my family for providing me with everything I needed when writing this thesis and during my studies.

Poďakovanie

Rád by som sa poďakoval vedúcemu práce doc. Ing. PhD., Michalovi Gregorovi za pomoc pri výbere témy, usmernenia pri písaní práce a konštruktívnu kritiku, bez ktorej by značne utrpela kvalita práce. Tiež by som sa rád poďakoval mojej rodine, za to, že mi poskytli všetko čo som pri písaní práce a počas mojich štúdií potreboval.

Abstract

This thesis is focused on designing an agent capable of controlling the gait of a hexapod robot. This agent was trained and tested in a simulated environment using the Soft Actor-Critic reinforcement learning technique. We describe the process of designing the simulated environment and neural networks that govern the agent's behaviour. We also present a guideline for transferring the obtained agent into the real world where it can control an actual robot. The simulated results show, that the agent was able to learn a gait that allows the hexapod robot to move forward. To a certain extent controlling the robot's movement direction is also possible. This work could in the future be used as a foundation for designing a model-free robot controller for different kinds of hexapod robots.

Keywords: robotics, deep reinforcement learning, artificial intelligence, simulation

Abstrakt

Táto práca sa zaoberá návrhom agenta schopného riadiť chôdzu robota typu hexapod. Tento agent bol trénovaný a testovaný v simulovanom prostredí s použitím metódy soft Actor-Critic, ktorá spadá pod metódy hlbokého učenia s odmenou. V práci opisujeme proces návrhu simulovaného prostredia a neurónových sietí, ktoré riadia správanie agenta. Predstavujeme tiež radu návrhov a usmernení pre prevod získaného agenta do reálneho sveta, kde môže slúžiť na ovládanie reálneho robota. Simulované výsledky ukazujú, že agent je schopný naučiť sa chôdzu, ktorá umožňuje, aby sa hexapod robot pohyboval dopredu. Je tiež možné, do určitej miery, ovládať smer pohybu robota. Táto práca by mohla byť v budúcnosti použitá ako základ pre navrhnutie bezmodelového ovládača pre rôzne druhy hexapod robotov.

Kľúčové slová: robotika, hlboké učenie s odmenou, umelá inteligencia, simulácia

Contents

List of Figures

List of Tables

Abbreviations

AI artificial intelligence, pp. 3, VIII

ALL Autonomous Learning Library, pp. 26–29, 33, VIII

DDPG Deep Deterministic Policy Gradient, pp. 8, 26

DQN Deep Q network, pp. 8, 11, 12, 17, VII

DRL deep reinforcement learning, pp. 2, 3, 26

GAE Generalized Advantage Estimation, p. 16

NN Neural network, p. 9

PPO Proximal Policy Optimization, pp. 14–17, 26

RL reinforcement learning, pp. 1–5, 7, 8, 16, 19, 23, 26, 29, 32, 44, 46

ROS Robot Operating System, pp. 2, 20, 44–46, 48, XII

SAC Soft Actor-Critic, pp. 8, 16–18, 23, 26, 28, 29, VIII

SGD Stochastic gradient descend, p. 9

TRPO Trust Region Policy Optimization, p. 15

UART Universal Asynchronous Receiver-Transmitter, pp. 46, 47

URDF Unified Robot Description Format, pp. 20–23, VIII, IX

1 | Introduction

In the last couple of decades, we have witnessed many groundbreaking developments that would have previously been reserved for science fiction. Modern mobile robots can walk, interact with objects and humans, even independently accomplish simple tasks previously only reserved for humans. However as the robots are becoming more complex by the year, the difficulty of developing them rises as well. This is most obvious with robots that utilize legs for locomotion, since this design forces engineers to develop complex controllers that allow such machines to move about. The design of these controllers is more often than not non-trivial and as such it is often reserved for experienced control engineers. Still, even with a lot of effort success is not guaranteed and even well-designed robots rarely achieve the same grace of movement that animals are capable of.

In the previous paragraph, we have essentially formulated the problem that this thesis addresses. For one way of solving it we can, as engineers often do, draw inspiration from nature. An animal learns to walk, not by performing a deep analysis of its tendons and muscles and then committing months to study books on control theory. No, an animal learns in a much more straightforward way by simply making attempts and attempting to achieve 'good results'. We write good results in quotation marks as measuring the goodness of something can be in practice a difficult and often very subjective task. Regardless, this simple concept of making attempts and obtaining a measure of success is, not only present in nature, but it is also at the core of one of the most promising sub-fields of artificial intelligence known as reinforcement learning (RL).

In this text we will focus on:

1. **Introducing deep reinforcement learning (DRL) algorithms**: with RL maturing as a field, a slew of different approaches at utilizing it becomes available. We will explore some of the most promising ones that are relevant for our use case.
2. **Simulation and modelling of a robot, task and reward**: just having an algorithm that works is not enough, so this chapter will be dedicated to creating a simulated environment within which we will teach a virtual robot how to walk.
3. **Implementing the DRL**: as its name suggests, this part of the thesis will be dedicated to detailed description of how we proceeded when implementing the DRL algorithm of our choice.
4. **Training and results analysis**: we will address the performance of the trained agent, suggest possible way of improving it and describe some of the difficulties we faced when running the training sessions.
5. **Transitioning into real world**: with the implementation and verification of the learning algorithm complete we will dedicate the last chapter to exploring the possibilities of transferring these results out from the simulation and into the real world. We will mostly focus on the Robot Operating System (ROS), which provides libraries and tools for managing communication and control of various types of robotic systems.

2 | Deep reinforcement learning

In this chapter, we will introduce concepts heavily utilized in DRL. These include, but are not limited to, state, action, value, policy, replay buffer, etc. Our goal here is to provide a common ground when discussing algorithms used in RL. Afterwards, we will focus on specific algorithms popular in the field, how they compare and how they differ from each other.

2.1 Introduction into DRL

Let's start by first examining the adjective *deep* that we have now used many times in conjunction with reinforcement learning. In the context of neural networks to proclaim that a network is deep, is the same as to say that it has multiple hidden layers, i.e. layers of neurons inserted between the input and output neuron layers. There is no specific set number of these hidden layers at which a network becomes deep, however, it is generally agreed that there should be at least two of these hidden layers in a deep network (see architecture in Figure 2.1 which has three hidden layers). In extension, deep reinforcement learning methods are considered deep, because they rely on the utilization of deep neural networks.

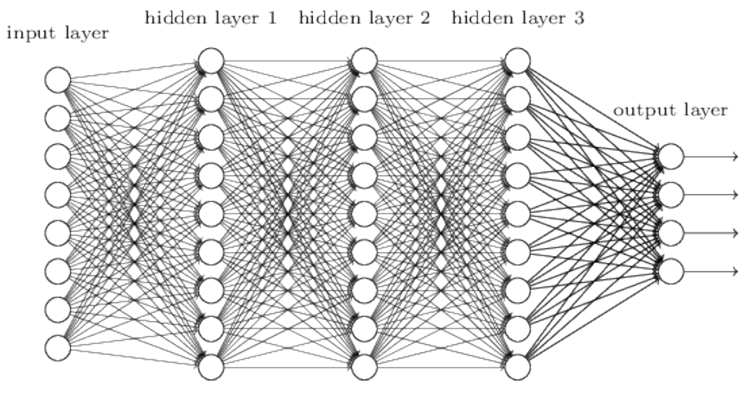


Figure 2.1: A deep neural network [1]

With this semantic issue out of the way, we can focus on exploring the various RL methods. Historically two major types of methods have been utilized in the AI. **Supervised** and **unsupervised** learning approaches. With supervised methods, an agent is trained with the help of existing data samples. These samples provide a clear example of what should be

LEARNING METHOD	**FEEDBACK**	**GOAL**
SUPERVISED	Correct output	Map inputs into correct outputs
UNSUPERVISED	None	Find hidden patterns in the provided data
REINFORCEMENT	Obtained reward	Maximize obtained reward

Table 2.1: Comparison of different learning methods

a *correct* response to the input. Because of this, a single data sample used for training has to contain input *and* output information. In contrast, unsupervised methods only work with the input data. In this data they attempt to find hidden patterns without being explicitly told what to look for.

The problem with both of these approaches is that they are not sufficient to address many of the problems engineers need to solve. Just finding a pattern in the input data doesn't tell the agent how to act on it. Also, we often find ourselves in a situation when we cannot obtain all examples of 'correct responses' to input stimuli. This can be caused by either technical difficulties in obtaining the data, the huge number of possible input-output combinations or our inability to even *tell* what the correct output should be like. The last case might be the very reason why we need the agent in the first place.

Think of the problem that we will be addressing in this thesis. How would one teach a robot to walk? An engineer could of course go through the drill of tuning controllers, solving inverse kinematics equations to construct a model and then obtain samples of 'correct' behaviour from that model. Once he had those he could use them to train a robot. But at that point, he would already have all he needed to make the robot walk anyway and the agent would not even be needed anymore.

Reinforcement learning very elegantly circumvents all these issues. No longer having to obtain input-output sample pairs, we can instead work with a reward obtained directly from the environment. A **reward** in this sense is just a number that gives the agent feedback on how well or how poorly it is doing. The goal of the agent is then no longer to try and mimic some "golden standard" as was the case with supervised learning. All it has to do is attempt to obtain as much reward as possible. This is the main idea that reinforcement learning is built on.

For further discussion on how the RL methods work, some terminology needs to be introduced. The first term we need to define is **state**, usually denoted as s. To put it simply, the state is everything that an agent receives as its input. The role of the state is to provide the agent with information about the environment it operates in. However, the choice of what information a state should include is not completely arbitrary. In fact, every state has to fulfil the so-called *Markov property*. In a mathematical sense, this property says that:

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_0, s_1, s_2, ..., s_t), \tag{2.1}$$

where t is used to denote a discrete time step.

Put in layman's terms: *The future is independent of the past given present* [2]. In theory, the agent should be able to make optimal decisions after observing the state. In practice, we cannot always ensure that this is the case (partially observable MDP), but we will not take this into account. We will refer to all states the agent can visit using the term *state space* (denoted S). Also, we will often abbreviate s_{t+1} to s' to keep the equations more concise.

Another important term is *action*, denoted a. Actions describe ways that an agent can interact with the environment. If the agent only has a finite number of choices it can make, we describe this as *discrete action space*. An example of this would be a chess-playing agent that can only make a set number of valid moves in each state. On the other hand, if the agent's output can be any real value in a given range (possibly in multiple dimensions) we speak of *continuous action space*. Since this thesis is concerned with an agent (robot) that operates in the real world, we will be most interested in continuous action space denoted A. Just like with state, we will also use a' as a short form of a_{t+1}.

We have already mentioned *reward* once before. While humans tend to associate the term with more tangible things, in terms of RL a reward is simply a number. This number provides the agent with feedback on its performance. We denote the reward r and obtain it from a *reward function* R. This reward function can in practice take different parameters in different contexts, for example:

- $R(s)$ is the reward agent obtains when entering state s
- $R(s, a)$ the reward obtained after performing action a in state s

- $R(s, a, s')$ is the reward obtained after performing action a in state s and entering a new state s'

The different way in which these are used will become clearer in the following section. It is also worth noting that the frequency with which the agent obtains a non-zero reward can vary significantly.

With these basic terms explained we can turn to more complex ones starting with the **return**, G. The return describes all reward accumulated during all time steps before the end of a training episode. Mathematically it can be expressed for infinitely long training episode as [3]:

$$G = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.2}$$

where the term γ was added to account for the uncertainty in obtaining future rewards and is usually picked from range $< 0.9; 0.99 >$.

Stated as in (2.2) the return is not that useful as it describes merely a single trajectory an agent can take during the episode. However, using it we can define a more useful property called the **value of a state** $V(s)$:

$$V(s) = E[G|s] \tag{2.3}$$

where E denotes the average (also called expected) return. We would obtain it if we collected data from many (ideally infinitely many) episodes and averaged it.

The value of a state is useful in providing the agent with an estimate of how good it is to be in any given state s. Still it doesn't tell the agent anything about how good/bad any of the *actions* it can make is. For that we define an additional property called *value of an action* Q(s, a):

$$Q(s, a) = E[r + \gamma V(s')] \tag{2.4}$$

(2.4) states that the value of taking an action a in the state s is the expected sum of reward r received after making the action and the discounted value of the reached state s'. It is also possible to define the value of the state in terms of Q and the value of action Q recursively:

$$\begin{aligned} V(s) &= \max_{a \in A} Q(s,a) \\ Q(s,a) &= E[r] + \gamma \max_{a' \in A} Q(s',a') \end{aligned} \tag{2.5}$$

The above equations (2.5) don't hold in general, but assume that both $V(s)$ and $Q(s,a)$ are optimal functions.

Lastly to describe the mapping from states the agents observes to actions it performs we define a policy π. π is defined as a probability distribution over actions, conditioned on the state:

$$\pi(a|s) = P[a|s] \tag{2.6}$$

The other way that policy π can be defined is to return the action with the highest probability of securing large reward in state s:

$$\pi(s) = arg \max_{a \sim A} Q(s,a), \tag{2.7}$$

When it comes to policy we recognize two main policy types, *deterministic* and *stochastic* policy. As the name would suggest, given identical input data a deterministic policy will always produce the same output. This is not always a desirable property as it can lead to a brittle policy. Stochastic policy, on the other hand, produces as its output probability distribution. The resulting action is then determined by sampling this distribution. This pushes the policy to better generalize. The equation (2.7) assumes deterministic policy.

With policy defined we can formalize the task of obtaining the most reward:

$$\theta^* = arg \max_{\theta} J(\theta) \tag{2.8}$$

(2.8) can be understood as: obtaining the optimal vector of parameters θ^* for a parametrized policy π_θ, to maximize criterion $J(\theta)$ [4], which itself represents the total expected return.

As the name of this thesis implies we will be discussing RL methods that primarily attempt to teach agent the policy. Even intuitively this makes sense. In the real world, we do not care about an arbitrary value assigned to various states we might encounter or actions we perform. From our point of view, the agent should just be able to tell what to do in any given situation. This is precisely what a policy is all about. The ultimate reason for using policy methods is that they work even in environments where A is continuous.

With the constant stream of new discoveries in the field of RL the number of available methods might seem overwhelming. It would not be feasible to mention every method that can be used to train a walking robot in this thesis. Because of that, we have limited our choice to three prospective candidates. These are Policy Gradient (DDPG), Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). These have been shown in past works [5][6][7][8] to be well-suited for our intended use.

Still, the distinction between value-based and policy-based methods is not quite as clear-cut as one might think, especially when it comes to actor-critic methods. For that reason and to fully appreciate the advantages and various nuances of the aforementioned policy-based methods it is necessary to also examine the value-based methods. When discussing these we will introduce multiple important concepts. Understanding them will make the theory behind policy-based methods easier to grasp.

2.2 Value based methods

As the name would suggest this class of methods relies on obtaining a good approximation for the state or action value functions and using them to drive the policy. Once these are obtained the agent can follow the trail of high-value states to obtain the rewards. A problem arises when either the state space or action space becomes too large to explore. This is where the neural networks finally make their appearance.

The problem of large state space has been alleviated by the use of a popular method combining the principles behind value-based methods with neural networks. The neural network takes the state as input and extracts important information out of it (effectively reducing the number of states). This ’condensed’ information gets passed to the next portion of the network, which learns to approximate the Q function.

This approach has been demonstrated to work well for example on agents playing old

Atari games and is known as deep Q network (DQN). Let's explore without much detail for now how this method is used in practice [9]:

1. Using the current policy π_θ populate a replay buffer with *experiences* from the environment
2. Randomly sample experiences from the replay buffer
3. Using the sampled experiences and *stochastic gradient descent* (SGD) update the NN to improve the Q function estimation
4. Repeat until convergence

Now is a good time to introduce the concept of a **replay buffer**. We have already made it clear that an agent learns from its experiences, but we have yet to mention that the experiences used for training must fulfil the so-called i.i.d. criterion. This is necessary to ensure that SGD works well. I.i.d. states that samples must be *independent and identically distributed*. The reasoning behind this is beyond the scope of this thesis, but it warrants the use of the replay buffer. This buffer contains the experiences $(s, a, r, s', done)$ obtained from the environment. By randomly sampling it we ensure that the agent is not being trained on closely correlated data. If this buffer was not included and we merely used the data obtained during a single episode, we would end up training the agent using temporally correlated data. Put a different way, adjacent steps that the agent makes in an environment don't accurately represent its overall behaviour.

As for the update of the neural network itself, we can perform that by trying to minimize MSE (mean squared error) between our target Q value and the current Q output [9]. This is also known as a *cost function* and we will denote it here as $L(\theta)$:

$$L(\theta_i) = E_{s,a\sim p(.)}[(y_i - Q(s,a|\theta_i))^2]$$
$$y_i = \begin{cases} r_T, & \text{if terminal state} \\ r + \gamma \max_{a'} Q(s',a'), & \text{otherwise} \end{cases} \tag{2.9}$$

Ideally, we want the $L(\theta)$ to be equal to 0. This would suggest that our Q estimate is in perfect agreement with the results obtained from the environment. Towards this end, we calculate the gradient of the loss function, which will tell us how to modify parameters θ. The resulting $\nabla_{\theta_i} L(\theta_i)$ is then multiplied by the learning rate and used to modify the parameters (network weights). The full equation to calculate this can be seen in (2.10). For visualization of the process see Figure 2.2.

$$\nabla_{\theta_i} L(\theta_i) = E_{s,a\sim p(.),s'\sim\varepsilon}[(r + \gamma \max_{a'} Q(s',a') - Q(s,a|\theta_i))\nabla_{\theta_i} Q(s,a|\theta_i)], \tag{2.10}$$

where the new state s' is obtained from the experience buffer ε.

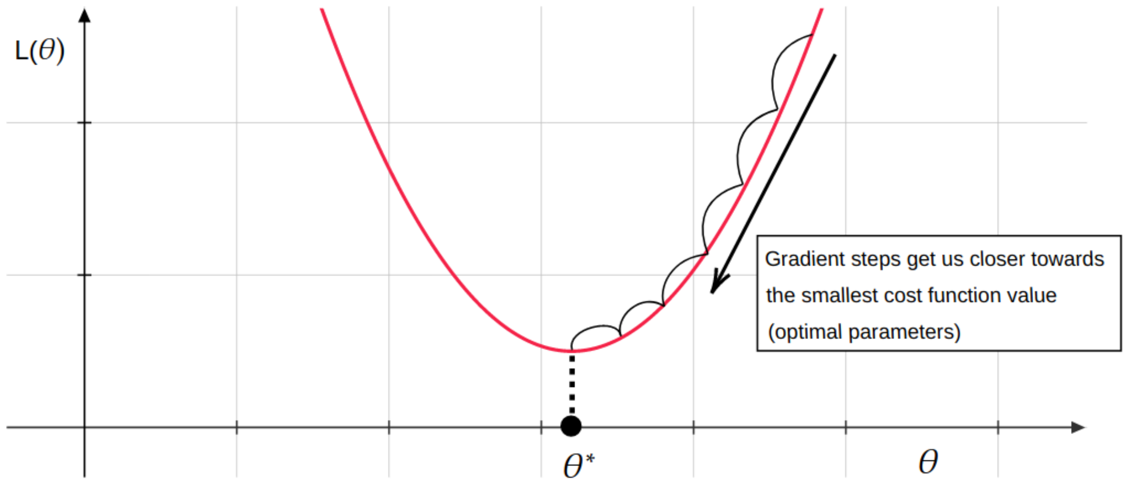


Figure 2.2: Visualization of gradient descent

The approach that was just described above does not always produce good results. One problem that arises is that because of the max in (2.9), the Q network is pushed to overestimate the action values. This destabilizes the learning and makes the policy behave chaotically.

A solution inspired by [10] has been proposed in [11]. It suggests the use of two Q networks in tandem. The target Q value y can then be expressed as:

$$y = r + \gamma \min_{i=0,1} Q_{\theta_i}(s', arg \max_{a'} Q_{\theta_1}(s',a')) \tag{2.11}$$

with the loss being defined for each Q network as in (2.12):

$$L(\theta_i) = (y - Q_{\theta_i}(s,a))^2 \tag{2.12}$$

As mentioned before the article [11] references [10], which itself is primarily concerned with solving the overestimation problem in actor-critic methods. Therefore in [11] the term $\pi_{\phi_1}(s')$ is replaced with $arg \max_{a'} Q_{\theta_i}(s',a')$ to reflect that in standard DQN methods policy is determined solely by action values. The author does not explain why he chose to use Q_{θ_i}

instead of Q_{θ_1} as the equations from [10] would suggest, so we resorted to using the latter form. In this thesis we will utilize this equation in the context of an actor-critic method and as such, we will be able to use its original form from [10] which includes $\pi_{\phi_1}(s')$. The form in (2.11) is listed because it can be utilized for standard DQN.

Estimating true Q this way, not only fixes the overestimation issue but also pushes the network towards lower variance states. We will see this trick used again later in actor-critic methods.

There is more that could be said about the DQN, but in terms of this thesis, this short introduction should be enough to provide the reader with a basic understanding of how it works. A similar approach as was just described for the Q network could also be used when using the state value – V network.

2.3 Policy based methods

We are now ready to introduce policy-based methods. An important distinction that has to be made clear is that there exist two major classes of methods. These are **on-policy** and **off-policy** methods. On-policy methods rely on the trained agent being supplied fresh data samples obtained from the environment. In this context, the word ’fresh’ means that if we are currently trying to improve the policy π_{θ_i} then for training we have to use data samples obtained *using this policy*! This requirement rules out the use of a replay buffer and in practice causes performance degradation (since the environment has to constantly supply new samples). This requirement does not apply to off-policy methods which can use a replay buffer just as in DQN.

2.3.1 Basic policy gradient

This method represents the most straightforward way of learning policy. While it has disadvantages that don’t make it a good fit for the problem we are addressing, it still makes for a good introduction to policy-based methods. A more exhaustive description of the method along with possible implementation in Python programming language can be found in [12].

Our goal using this method is to maximize the expected return $J(\pi_\theta)$. The agent receives this reward from environment following the trajectory τ obtained using π_θ. Put formally:

$$J(\pi_\theta) = E_{\tau \sim \pi_\theta}[R(\tau)] \tag{2.13}$$

For optimization of the parameters we want to use gradient ascend (which is in principle very similar to gradient descent as seen in DQN). The form of the parameter update should therefore be:

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta J(\pi_\theta | \theta_i) \tag{2.14}$$

Finding a way to estimate **policy gradient** $\alpha \nabla_\theta J(\pi_\theta$ is crucial not only in this method but also in many other policy gradient-based methods. The first step is to write the expression for calculating the **probability of trajectory** given some policy parameters:

$$P(\tau|\theta) = P(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \tag{2.15}$$

When deriving the gradient of this probability we can effectively utilize the *log-derivative trick*. Realizing that derivative of $\log x$ with respect to x is $1/x$ we can write:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) \tag{2.16}$$

With this, we still need to express the $\nabla_\theta \log P(\tau|\theta)$ in terms of the policy. We do this by rewriting the (2.15) in terms of logarithms and taking the gradient of both sides. In [12] we can see the result of this. Multiplications were replaced by summations (since $\log x + \log y = \log xy$) and the crossed-out terms were taken out since they were not dependent on θ and as such their gradients were zeros.

$$\begin{aligned} \nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log P(s_0)} + \sum_{t=0}^{T} (\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t)) \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \end{aligned} \tag{2.17}$$

With this we can return back to the original problem we were addressing and describe the policy gradient in terms of policy and reward:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \nabla_\theta E_{\tau\sim\pi_\theta}[R(\tau)] \\ &= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\ &= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau)\end{aligned} \tag{2.18}$$

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)] \tag{2.19}$$

Since the gradient is determined in terms of the expectation we can obtain it by averaging across the set of trajectories sampled from the environment. Here is where one of the disadvantages of this method becomes apparent. To update the parameters, we are using returns obtained over the entire episode and while this provides us with a good estimate of how the agent did overall, it does not specifically reward or punish individual actions. This hints at long training times which is indeed the case in practice. It also causes another, slightly less obvious problem. If we took an action at time step t it is obvious that this action a_t did not influence rewards $r_{t-1}, r_{t-2}, ...$, yet this is not taken into account.

A modified version of the policy gradient can alleviate this issue:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= E_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{R}_t] \\ \hat{R}_t &= \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})\end{aligned} \tag{2.20}$$

Using this form of gradient the actions will be no longer be reinforced based on rewards they didn't contribute to. This shows that by manipulating the $R(\tau)$ part of the (2.19) we were able to improve the gradient policy update. We can take it one step further when we utilize the lemma in (2.21). This lemma holds for any parametrized probability distribution P_θ over a random normally distributed variable x.

$$E_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0 \tag{2.21}$$

The proof of this lemma can be found in the appendix (Expected Grad-Log-Prob Lemma) or also in [12]. With this lemma in mind we can write:

$$E_{a \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) b(s)] = 0 \tag{2.22}$$

This equation states that since we know that the original expectation has the mean at 0, multiplying it by a baseline (state only dependent) function $b(s)$ will not change the expectation. A good baseline has the potential to decrease variance in the obtained data and improve learning.

In practice the most popular baseline is called **advantage**, denoted here as $A_{\pi_\theta}(s, a)$. Intuitively it can be understood as a measure of how much better it is to take action a in state s than it is to take an 'average' action. The advantage is defined as:

$$A_{\pi_\theta}(s, a) = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s) \tag{2.23}$$

Using the advantage as a baseline improves the learning, but requires additional work, as it forces us to estimate the action *and* state value. This can be done with the utilization of value-based methods mentioned in the previous section. We will also briefly discuss this issue in the next section when describing PPO.

2.3.2 Proximal Policy Optimization (PPO)

In the previous section, we described how a policy can be improved by finding gradients and taking steps towards the optimal policy (π^*). The problem with this approach is that even a small change we make to the policy can significantly change the agent's behaviour. This forces us to use a small learning rate α, which results in small steps and slow learning. If we were to chose larger steps the training process could become noisy or might not converge at all.

This problem was the motivation behind PPO and TRPO methods which attempt to address it. They do so by monitoring the 'distance' between the old and new policy to ensure that these never become too dissimilar. If that were to happen it could hamper the learning process. But while both of these methods address the same problem, they do so in different ways. Since the PPO is generally accepted as easier to implement while matching the performance of TRPO we decided to focus only on PPO in this thesis.

Just like the vanilla policy gradient, PPO (and TRPO) is also an on-policy method. It can also be used with both discrete and continuous action spaces. The PPO however updates policies differently [13]:

$$L^{CLIP}(s, a, \theta_k, \theta) = min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\mathrm{A}^{\pi_{\theta_k}}, clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon), (1+\epsilon)\right)\mathrm{A}^{\pi_{\theta_k}}\right) \tag{2.24}$$

Note that in (2.24) $\pi_{\theta_k}(a|s)$ represents the old policy and that the advantage $\mathrm{A}^{\pi_{\theta_k}}$ is a function of s, a (omission only done in order to simplify the notation).

To encourage exploration entropy can also be added to (2.24). This works because the reward from high entropy pushes the agent to behave more randomly (a lot of exploration), while the reward for good performance drives it to improve the actions it makes. Using this the final loss for the actor is [14]:

$$L(\theta) = L^{CLIP} + \alpha H[\pi_\theta](s) \tag{2.25}$$

where α is a hyperparameter that sets how much importance we attribute to the entropy.

Since PPO is categorized as an actor-critic method we need to train two models when utilizing this method. The actor is responsible, as the name would suggest, for choosing which action to take in any given state. From this, it is obvious that it is the actor that will dictate the resulting policy π. Critic, on the other hand, plays a role in estimating the advantage function, either through action value or state value. In practice it is also possible to have two critics, one estimating the state, the other action value. Training these networks in PPO is no different from training them when they are used on their own. If we were calculating the advantage using the value function we could use a loss function formulated as [14]:

$$L^{VF}(\theta) = (V_\theta(s) - V_{REAL}(s))^2 \tag{2.26}$$

We have already shown that one popular method of computing values is using the discounted return as defined in (2.2). There is also another popular method often used with PPO. It is called Generalized Advantage Estimation (GAE). GAE provides a convenient and easy-to-compute way of smoothing the state values, which in turn smooths out the advantage and improves the learning performance. When using the GAE to estimate returns we follow a straightforward procedure:

1. Set initial GAE value as 0
2. If a value was obtained at terminal state assign it mask=0 otherwise mask=1
3. Starting from the last step compute $\delta = r + \gamma V(s').mask - V(s)$
4. $GAE = \delta + \gamma\lambda.mask.GAE$
5. $V_{REAL}(s) = GAE + V(s)$

The parameters γ and λ are hyperparameters that are usually about 0.99 and 0.95 respectively.

2.3.3 Soft Actor-Critic (SAC)

We saw in the previous section that PPO offers a relatively simple to implement and robust way of training policy based agents. Because of that, it might seem that there is no need for us to look for other RL method, but as was mentioned PPO has one weak side. That weak side is sample efficiency, or rather inefficiency. This stems from the fact that PPO is an on-policy method and all training has to happen on fresh data samples. This forces us to either constantly run the simulation (high computational demands or long learning time) or to obtain new data samples from the real world (impractical and very lengthy).

Soft Actor-Critic (SAC) resolves this issue as it falls under the off-policy methods. This significantly improves its sample efficiency. Samples, once obtained, are pushed into a buffer and can be used for training even at a later time, reducing the frequency at which we have to obtain new samples from the environment. Already there are examples of the application of SAC in the real world to train walking robots in a reasonable amount of time [5]. From this, we can presume that if we use SAC to train the hexapod robot, considered in this thesis, we should be able to move from the simulated environment into the real world with few issues.

Let's address the inner workings of SAC more in-depth. Just like PPO it is an actor-critic method (implying we have to train at least two networks). Its main goal is that it tries to maximize the expected return between obtained reward and entropy. With the entropy:

$$H(P) = E_{x \sim P}[-\log P(x)] \tag{2.27}$$

The optimal policy π^* can be represented as:

$$\pi^* = arg\max_{\pi} E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t))\right)\right] \tag{2.28}$$

where the temperature coefficient α determines how much importance is attributed to the entropy.

This describes what we are trying to accomplish using SAC, but it does not tell us how to do it. For that we need to define loss functions at least for Q and π networks. Starting with the Q networks, it is important to note the plural form. Indeed we will need to utilize two Q functions. The reasoning behind this was already explained in the DQN section. Both of these Q functions however can be trained the same using the target:

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\odot},i}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \tilde{a}' \sim \pi_\theta(\cdot|s') \tag{2.29}$$

where $d = 1$ if s is a terminal state and as noted $\tilde{a}'$ is obtained from the current policy.

With this target defined we can update both Q networks using simple MSE loss:

$$J_Q(\phi_i, D) = E_{(s,a,r,s',d)\sim D}\left[\left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2\right] \tag{2.30}$$

We will also note here (and return to this point later when discussing the implementation of SAC in our code) that (2.29) is not the only way to compute the loss. This equation can be further broken down if we also add a network for estimating state values $V(s)$. In that case, the expression in the brackets would become the target for the value network allowing us to replace the contents of the brackets with $V(s')$.

Deriving the loss function for the policy network can be quite cumbersome as it relies on the reparametrization trick and some simplifications of the resulting formula. Instead of listing the mathematics [15–17] here we only provide the resulting loss function:

$$J_\pi(\theta) = E_{s\sim D}\left[E_{a\sim\pi_\theta(\cdot|s)}[\alpha \log \pi_\theta(a|s) - Q_\phi(s,a)]\right] \tag{2.31}$$

With both loss functions now defined, we could already implement the SAC as it was first described in [16]. But there still is one more improvement we can do. Up to this point, we have mostly ignored the temperature hyperparameter α which determines the overall 'importance' of the entropy. As it turns out, it is quite sensitive and can either help or significantly jeopardize the learning. It is also practically impossible to find a value of α that works well for multiple different problems. Because of these issues, most modern implementations of SAC make use of an automatic scheme for adjusting the entropy temperature. This is usually done by finding entropy loss and taking its gradient. The loss function is:

$$J_\alpha = E[-\alpha \log \pi(a|s;\alpha) - \alpha\bar{H}] \tag{2.32}$$

where the $\bar{H}$ is the preferred minimum entropy usually set to a zero vector.

3 | Simulation and modelling of a robot, task and reward

All reinforcement learning methods mentioned in the previous chapter have one thing in common. They rely on information sampled from the environment. In the context of the problem we are addressing in this thesis, there are two ways to obtain these information samples. One is to construct the robot we are teaching to walk and work within the confines of the real world, the other to use a model of the robot and the environment. Both of these have their pros and cons.

The main problem when dealing with a real, tangible robot is the rate at which we can obtain the samples. Even relatively simple RL problems might require millions of samples to achieve any level of success. In the real world, this would translate to dozens or even hundreds of hours spent on training. During this time the robot might also get damaged or special testing rigs might have to be manufactured. Despite these disadvantages learning in the real world offers many advantages too. One is that there is no need to model the robot or the environment, a task which in some cases might be exceptionally difficult on its own. Another advantage is that once the agent is trained no more additional work is required to implement it.

The other, often much more preferred way, is to use a computer simulation. This way the learning rate becomes constrained only by the hardware used and thus a much larger selection of learning methods becomes available. Another advantage of simulated environments is the perfect control over the robot and the environment and the lower costs associated with not having to construct anything. Training agents in simulated environments doesn't come without its issues though. For one the model used will always be a simplification of the real thing and as such there are no guarantees that what works in a simulation will also work in reality. Also as was mentioned previously, constructing a reasonably accurate model might be a difficult problem on its own in some cases.

We find ourselves in a position where we could attempt both approaches, since the robot we are considering in our learning problem already exists. We decided however that the flexibility the simulated environment provides suits our needs much better. Because of this, we will be training the agent in a simulation.

3.1 Modeling the robot

One of the most popular ways to model a robot is a format called URDF (Unified Robot Description Format). This is an XML based format, which represents a robot as a set of bodies connected through links. These links can represent either fixed connections or various types of joints such as revolute, linear, spherical, etc. Joints can be given additional properties that allow them to be treated as motors with set maximum torque, speed, power, etc. This allows for different modes of control and a good approximation of the real hardware.

Bodies that are linked by these joints can be defined either by simple geometrical shapes (so-called primitives) or by more complex meshes composed of hundreds of triangles. In this case, a compromise has to be made between the accuracy of the bodies and the overall performance of the model. Meshes are in general significantly harder to model, therefore primitives should be used whenever possible. Besides their shape bodies can also be given additional properties which depend on what purpose the individual body is supposed to serve. A body can either serve as a purely visual representation of the robot, in which case we could define its colour and texture. But the body can also be used to define features such as weight distribution and collision maps, that determine which parts of the robot will be able to exert force on the environment (or on other bodies).

Another added advantage of the URDF is that it is supported by Robot Operating System (ROS for short). Thanks to this any URDF model can be integrated into ROS's suite of simulation tools, which then makes it easy to develop and test a deployable code on a real robot.

To construct a URDF a few things must be known about the robot beforehand. How many motors does it have, how are they utilized to provide locomotion, how large is the robot, etc.

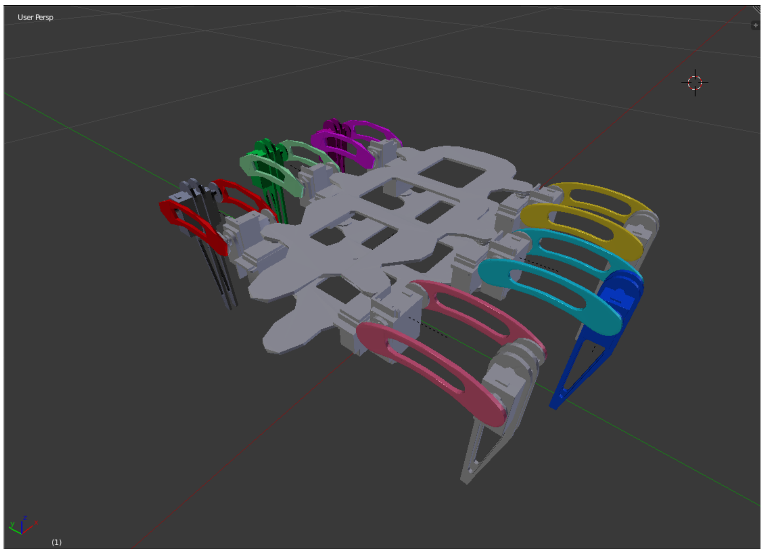


Figure 3.1: Model files that served as a starting point for URDF design

In our case, this work has already been done by a different team. As such we were provided with the graphical model of the robot constructed in a Blender project format. This model provided a starting point but was by itself not usable for any simulation environment.

While it would be theoretically possible to construct the entire URDF in any text editor, this approach would take far too much time with a robot as complicated as the one we intended to model. Instead of writing the model directly, we took advantage of a community of Blender users which has produced a multitude of add-ons for the software. From all the available packages, we selected an older one called Phobos that implements useful URDF-related functionality. [18] Phobos allows users to define the URDF directly from within the Blender environment.

To better explain how a URDF model is constructed we can refer to the Figure 3.2. Within the URDF file, bodies are referred to as links and it is within these links that the visual and physical properties of the body are defined. These links are connected by joints to form a tree-like structure with a single link defined as origin. Our URDF file will have to follow the same structure.

Phobos makes it easy to define the visual and collision geometry of individual links. Standard Blender features then allow us to reduce the number of polygons in a given mesh. We utilized this feature to simplify the meshes in order to avoid having to replace them

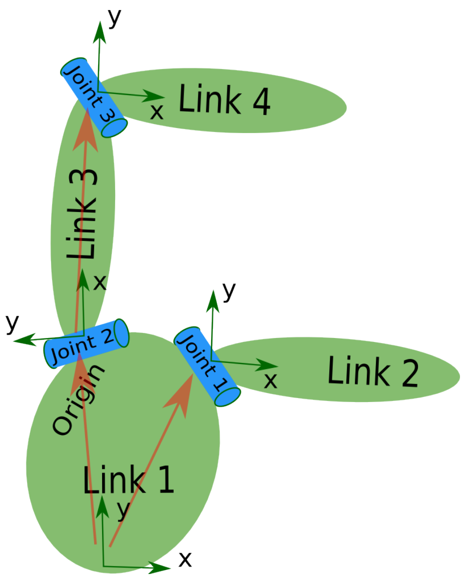


Figure 3.2: In URDF links are connected in a hierarchy with individual links being connected by joints [19]

with primitives. This kept the model more accurate and will later improve simulation performance. The only exceptions were the six servo arrangements connected to the base body of the robot. These were simplified to primitive cubes due to pybullet import issues. All 18 joints connecting the robot's legs were defined as simple revolute joints. These were connected to the base link using two rigid links, giving us a total of 21 joints, 18 of which can be controlled. The final step before exporting the finished URDF was to add a mass and inertia to all simulated bodies. The final model can be seen in Figure 3.3.

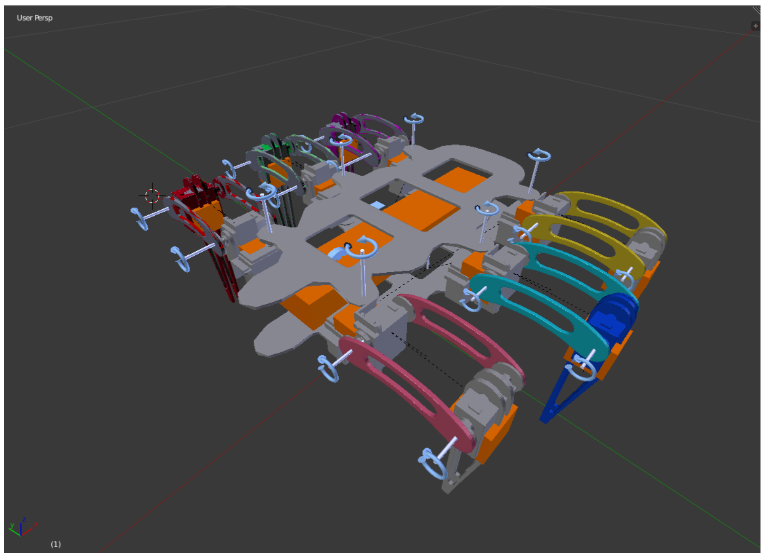


Figure 3.3: Finished Blender URDF model, with visible joints, links and masses (displayed as orange boxes)

With these steps completed the model can now be exported into URDF. This process creates a directory with <>*.stl* files (one for each utilized mesh) and a separate text-only <>*.urdf* file. This URDF file contains all the specified information and references to the .stl files.

3.2 Simulating the environment, task and reward

With the URDF file complete a physics simulator engine is needed. It has to be capable of both utilizing the URDF file format and also facilitating the necessary physics simulation. Our simulator of choice was pybullet. Pybullet is available for free and has already been utilized for simulation in many other robotics applications.

Once downloaded and installed pybullet can be used from within any python script in either GUI or DIRECT mode. In the GUI mode pybullet also renders the environment. This feature allows a user-friendly examination of the simulation (at the cost of worse performance).

Pybullet can provide the physics simulation, rendering and control over the robot and its state, it can't however help with the RL problem. For that, we will need to use another python module named *gym*. Using a gym environment will allow us to formulate the AI agent in a way that will allow for easy access to the state, action, reward and all other information and functionality we will need during the training. Use of the gym module with its *Environment* class has become a de-facto standard way of providing an interface to RL agents.

Thanks to the work of pybullet's creators the module also includes classes that make working with various importable types of robots easier. The one suited for our use is called *URDFBasedRobot*. It includes methods for easy import of URDF models and encapsulates all joints and bodies into sub-classes. These subclasses provide additional methods for control and information gathering. How all these individual pieces fit together and interact can be seen in Figure 3.4.

With the ability to simulate the environment taken care of we needed to decide on how to represent the state s. Examining already proven approaches [5], we decided that the state representation s had to include: joint angles (α), joint velocities (ω), yaw, pitch and roll

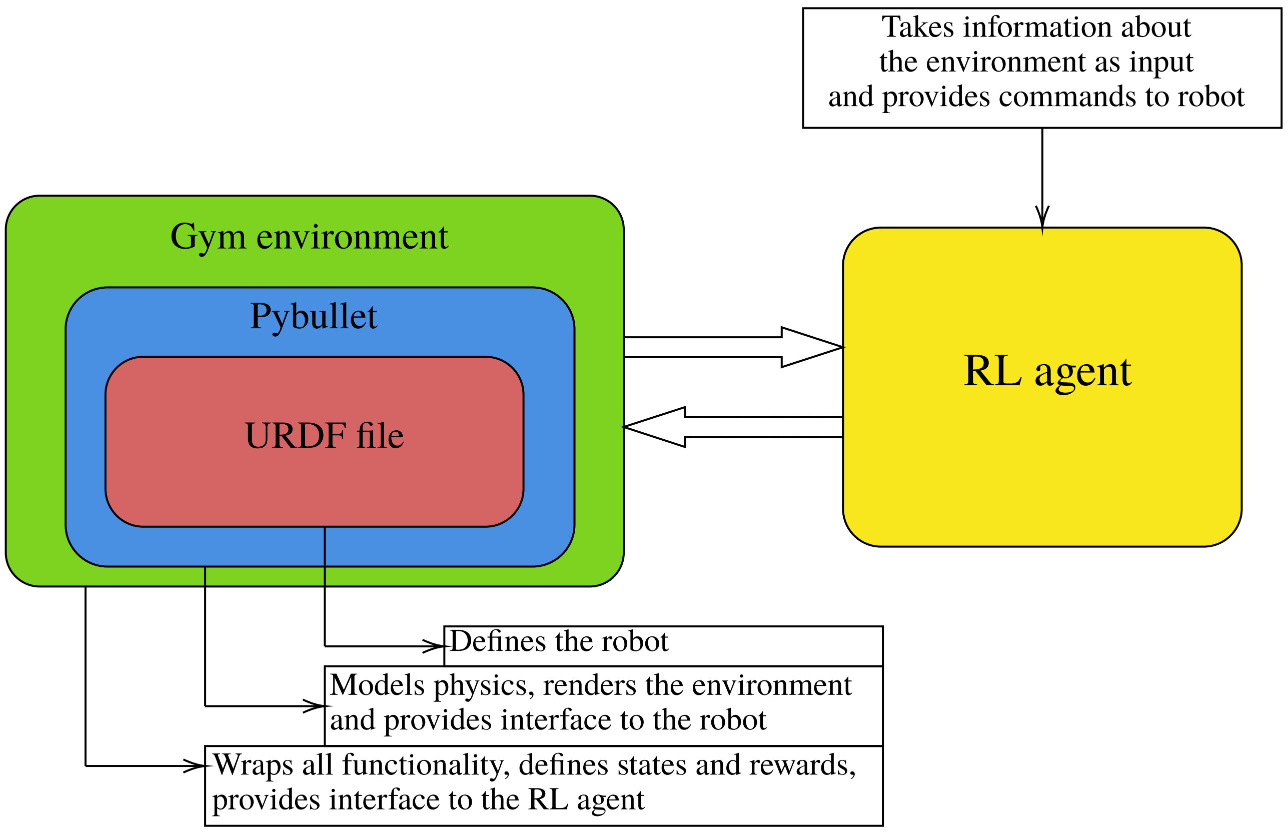
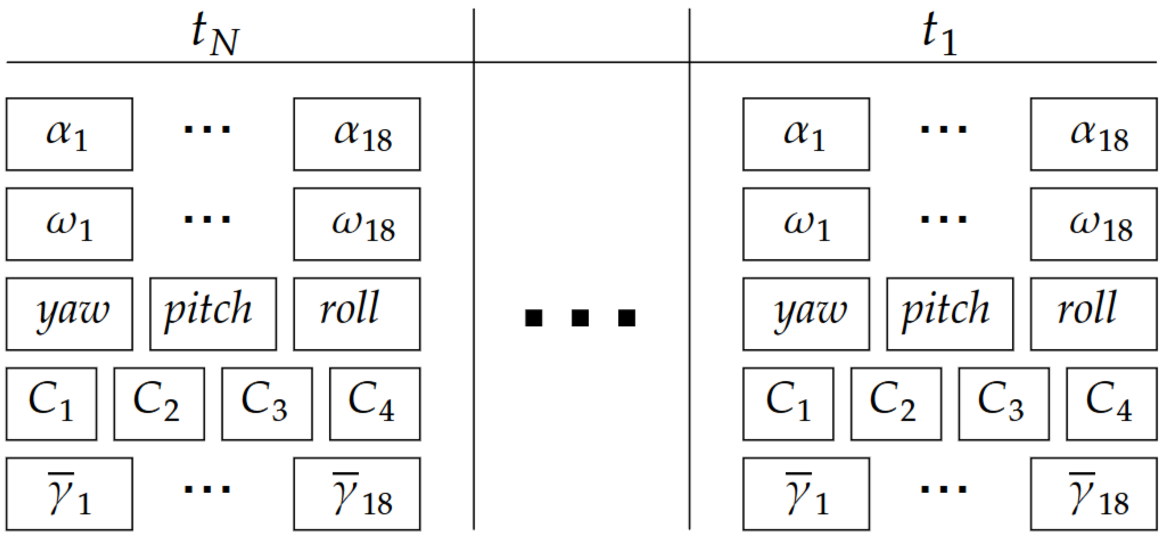

Takes information about
the environment as input
and provides commands to robot
Gym environment
Pybullet
URDF file
RL agent
Defines the robot
Models physics, renders the environment
and provides interface to the robot
Wraps all functionality, defines states and rewards,
provides interface to the RL agent

Figure 3.4: High-level overview of the code structure and interaction between the SAC agent and the gym environment



t_N
t_1
α_1 ⋯ α_{18}
ω_1 ⋯ ω_{18}
yaw $pitch$ $roll$
C_1 C_2 C_3 C_4
$\bar{\gamma}_1$ ⋯ $\bar{\gamma}_{18}$

Figure 3.5: State representation

of the robot and four control bits ($C_1 - C_4$) that determine the robot's movement. This is however not enough as the task formulated this way exhibits very strong non-Markovian properties [5]. To alleviate that we used a similar trick as Berkley scientists and extended the state with the history of previous N observations and actions ($\bar{\gamma}$). We decided to keep N as a hyperparameter, so we could examine how changing it impacts the performance. The resulting state representation can be seen in Figure 3.5.

4 | Implementing DRL

After considering the available RL methods we decided to go with the SAC. The choice was simple as SAC offers the combination of offline learning and the availability of ready-made implementations that allow us to focus solely on the task at hand. The implementation that we chose and introduce in the following section is called Autonomous Learning Library (ALL) [20].

4.1 About Autonomous Learning Library (ALL)

In the documentation of ALL, its authors describe it as:

"The autonomous-learning-library is an object-oriented deep reinforcement learning (DRL) library for PyTorch. The goal of the library is to provide the necessary components for quickly building and evaluating novel reinforcement learning agents, as well as providing high-quality reference implementations of modern DRL algorithms."

The library can be used in combination with Python3 programming language and as the above quote states, it relies on the use of PyTorch. For completeness, we will state that PyTorch and its alternative Tensorflow are two of the most popular deep learning libraries, used for constructing and training neural network agents. In this case, the ALL calls on methods supplied by PyTorch and hides the complexities of the learning algorithm from the end-user.

Besides the SAC, ALL also implements other DRL algorithms such as PPO, DDPG, Policy Gradient, etc. On top of that ALL is also very good when it comes to separating the learning algorithm, neural network design and the task being solved. This modularity is illustrated in Figure 4.1 bellow:

As can be seen, ALL only requires that the environment it is supplied is of the gym or gym-like type and it can derive the observation space and action space accordingly. This allows for the environments to be easily modified independent of the RL agent and even for the task to be changed without the need to modify the agent itself in any way.

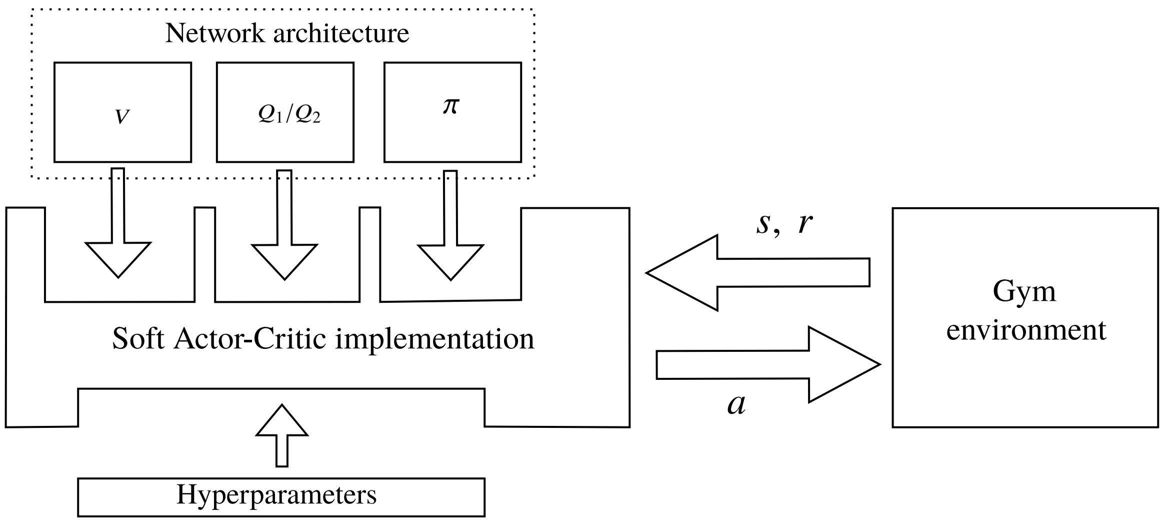


Figure 4.1: ALL provides interface that promotes modular approach to designing RL agents

In this sense, the only disadvantages that ALL presents to the end-user is that as a relatively new library it still lacks some features. One such shortcoming is the cumbersome way the pre-made implementations handle logging data, which on default settings creates unreasonably large log files. Another feature missing in ALL is the ability to easily load trained networks (while it may require adding in user code). We addressed both of these issues in a relatively straightforward way by extending the library with our code.

When defining the network architecture ALL provides a handful of premade classes and example files that showcase how to use them, this can be seen in Listing 4.1. On line 1 it can be seen that changing the number of hidden neurons is a simple matter of modifying the hidden1, hidden2 parameters. Additionally, layers can be added or removed in the body of the *fc_q* function. Architectures for value and policy networks are defined in a very similar fashion.

```
def fc_q(env, hidden1=400, hidden2=300):
   return nn.Sequential(
      nn.Linear(env.state_space.shape[0] + env.action_space.shape[0] + 1, hidden1),
      nn.ReLU(),
      nn.Linear(hidden1, hidden2),
      nn.ReLU(),
      nn.Linear0(hidden2, 1),
)

q_1_model = fc_q(env).to(device)
q_1_optimizer = Adam(q_1_model.parameters(), lr=lr_q)
q_1 = QContinuous(
   q_1_model,
   q_1_optimizer,
   scheduler=CosineAnnealingLR(
      q_1_optimizer,
      final_anneal_step
   ),
   writer=writer,
   name='q_1'
)
```

Listing 4.1: Example of a Q network with two hidden layers of 400 and 300 neurons

The Listing 4.1 also shows that the ReLU activation function and Adam optimizer can be potentially swapped if the agent's design required it. Finally, we observe that we do not need to define the input size (the output of Q is always just 1) since it can be derived from the data contained in the environment itself.

4.1.1 Implementation of SAC in ALL

We have already outlined in 2.3.3 how SAC operates in the general sense, but let's now more closely examine how exactly our library of choice implements it. As was mentioned previously the (2.29) can be further broken down if we also decide to use value estimating neural network. This is exactly what ALL does and as such the new target for Q turns from (2.29) to just:

$$y(r, s') = r + \gamma V_{\odot}(s') \tag{4.1}$$

The (4.1) while simple still needs some explanation. For one we no longer need to know whether the step we are considering was the final one since ALL doesn't save final steps into the experience buffer. Also, the $V_{\odot}$ was used because, to improve the stability of learning, ALL implements a so-called target network. What this means is that instead of having just one value network we update and get values from, we now have two networks:

- *base network* (V_B) - updated frequently, values from it used in J_V
- *target network* ($V_\odot$) - periodically synchronized with V_B, values from it used in J_π

Using this approach we avoid the problem when changes in value network lead to vastly different gradients at each training step. The V network itself then uses the target which it replaced in (2.29) and we get the loss function:

$$J_V(\xi_i) = \left(V_B(s) - \left(\min_{i=1,2} Q_{\phi\odot,i}(s,\bar{a}) - \alpha \log \pi_\theta(\bar{a}|s)\right)\right)^2, \bar{a} \sim \pi_\theta(\cdot|s) \quad (4.2)$$

4.2 Architecture of the RL agent

With these differences in implementation explained we can now look at the changes we made to the network architecture. Once again we used [5] as a reference when designing the architecture. The team behind Minitaur used a neural network with two hidden layers. Each of the hidden layers contained 256 hidden neurons, with the optimizer used being ADAM (learning rate $3 \cdot 10^{-4}$) for both Q estimators and their policy estimator. The entropy learning rate was likewise set to $3 \cdot 10^{-4}$ with the target entropy being equal to $-N_{JOINTS}$.

In [5] it was shown that the network designed for Minitar could also be successfully applied to other problems. We however decided to increase the number of neurons in hidden layers twofold, because of the significantly larger observation space of our problem. For comparison, the observation space of Minitaur consisted of only 112 values while the hexapod robot with the same number of previous observations on input has 348 values. All activation functions used were ReLu.

The ALL's example implementation of SAC included also cosine annealing learning rate scheduler [21]. This works by starting with a large learning rate which is over time lower and then increased again to simulate a "warm restart" of the learning process. We left this part of the implementation untouched. Finally, the minibatch size, replay buffer size and initial buffer size for replays were all also left unchanged at 100, 1e6 and 5000 respectively.

With the network architecture set up this way another thing to note is we obtain values from the π estimator. The most straightforward way might seem that with N outputs for N joints the output dimension should also be N, but in fact, the output dimension is $2N$. This is because with a single number we run a significant risk of overfitting the network while making it very sensitive to small changes. Since we desire a robust policy, it is much better to obtain each output by sampling a Gaussian distribution, where each joint is represented by *mean* and *variance*. When done this way small changes in the output still lead to relatively similar output distributions.

The final problem that needed to be solved was the design of the reward function. The core idea of the reward function we ended up using was taken from the pybullet's implementation of Minitaur (which differs from the reward function in [5]). Its core idea was that the agent obtains reward only when the robot progresses past the point it occupied in the previous step R_0. All other behaviour of the agent is then moderated by punishments it gets for poor performance. These punishments tracked three criteria:

- sideways movement (R_1) - the robot's movement should be constrained to X-axis so any change in the Y-axis should be punished
- rotation of the body against the Z-axis (R_2) - motivates the robot to avoid "bouncing around"
- energy expenditure (R_3) - calculated as $\sum_{i=1}^{N_{JOINTS}} \omega_i * \tau_i * \Delta t$, where ω_i is joint velocity, τ_i joint torque and Δt is time between steps

The final reward obtained in one step is then:

$$R_{tot} = w_0 R_0 - w_1 R_1 - w_2 R_2 - w_3 R_3 \tag{4.3}$$

, where $w_0 - w_3$ are parameters which set the relative weights of individual reward components. These relative weights can be obtained through random search of the parameter space. In UNIX environment this search can easily be accomplished using a simple *bash* script. This search produced the final values $[w_0, w_1, w_2, w_3] = [10833.473; 0.81; 65.764; 18.75]$.

The very last thing that is needed is a criterion for terminating an episode. One of these criteria is that if the robot's body exceeds 90° roll angle the episode is considered failed and as such is terminated. If this happens we also supply a large negative reward, which only reflects in the total reward obtained during the episode and has no effect on learning (since the last steps are not added into the experience buffer). Another condition that ends an episode is exceeding a time limit.

More on why using a set time limit is not the best approach will be mentioned in the next chapter. In the end, we determined that the best results can be achieved if besides a fixed time limit we also use a *"death-zone"*, which approaches the robot at a fixed speed. This way initial episodes can be much shorter than the maximal episode time limit and as the robot learns to walk better it can get away from the death zone at sufficient speed to "survive" until the episode gets terminated by the time limit. An episode can also be terminated if the robot reaches the edge of the ground plane.

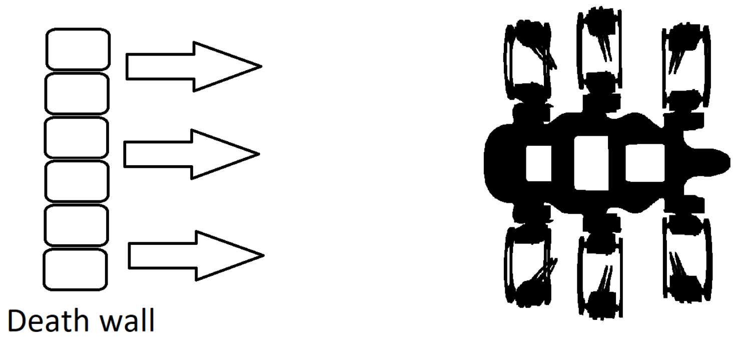


Figure 4.2: Task visualization

5 | Training and results analysis

With the theory and design of the RL agent's architecture explained we now move on to describe the learning process.

5.1 Running training and testing network architecture

The codebase for running the training was created to be easy to use and install. As such the code includes a bash based install script for UNIX based operation systems which installs are the prerequisites, Nextro gym environment and the pytorch library needed to run the training. It also installs tensorboard software, which is very useful for logging and displaying information related to the learning process such as loss, reward, mean reward, max reward, temperature, etc.

In order to start locate the file *agent_minitaur_inspired.py* and in the directory run:

```
python agent_minitaur_inspired.py --mode=train --frames=1000000
```

Listing 5.1: Running the training over 1000000 frames with default settings

This will start the training process, with pybullet in direct mode (no visualization). The training will be considered finished after 1000000 frames were processed. After every complete training episode user will be informed about the total collected reward, FPS (frames processed per second) and the total number of processed frames. Every 10 episodes the weights of all networks will be saved and when the training finishes user-tunable settings (such as $w_0 - w_3$ coefficients) will also be saved. The data available after a full course of training is:

- events.out.tfevents.<...> = tensorboard data
- policy.pt, q_1.pt, q_2.pt, v.pt = network weights
- settings.p, settings.txt = information about the user settings (in plaintext and pickle form)
- 100.csv = information about the rewards obtained during training

First we take look at the results obtained from the agent, with all settings as described in the previous section. The used hyperparameters were obtained through random selection with this specific result being shown, because with these settings the agent was able to learn how to walk forward using all six legs. The command used to train this agent was:

```
python agent_minitaur_inspired.py --mode=train --frames=1500000 --rew_params 10000 0.05 10 10
```

Listing 5.2: Running the training with hyperparameters $(w_0, w_1, w_2, w_3) = (1e4, 0.05, 10, 10)$

Training took approximately three hours, but of the 1500000 frames only one in every two was used to facilitate learning (this was the preset value that programmers of ALL chose). Results obtained during the training run and displayed with the use of tensorboard are shown bellow in Figure 5.1:

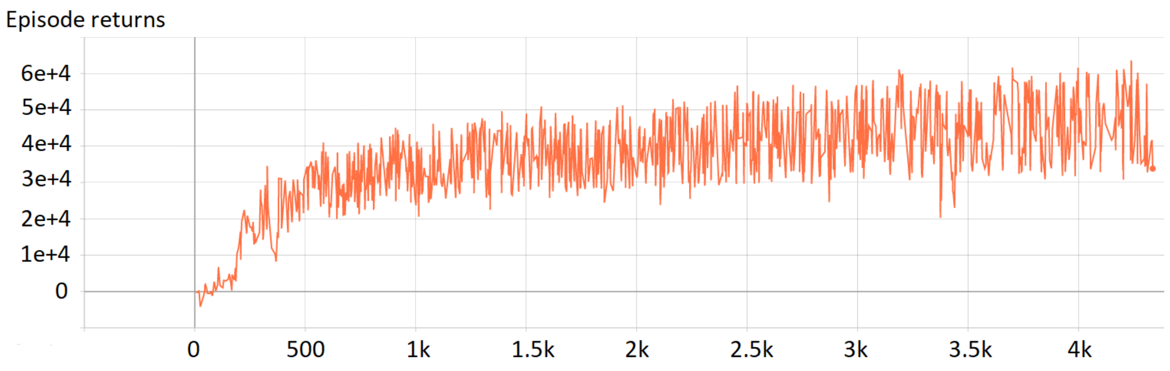


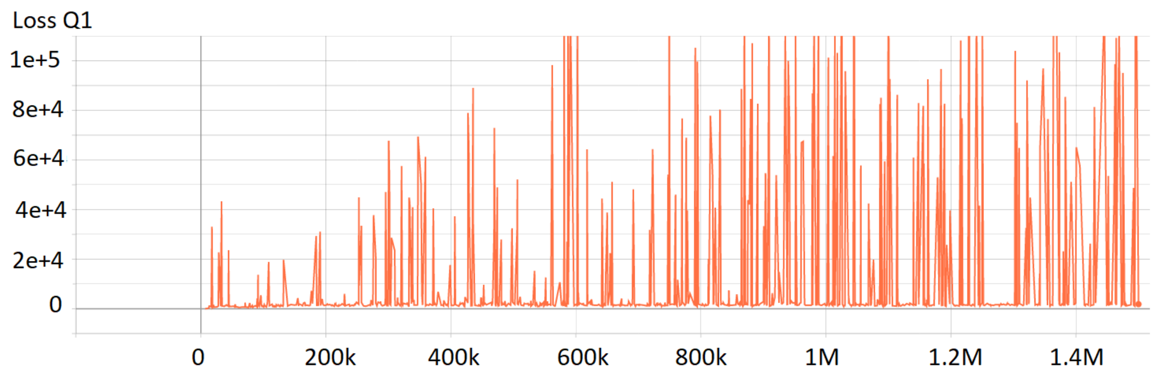
Figure 5.1: Reward collected

The steadily rising mean reward clearly shows that the agent was capable of learning. When the behaviour of the agent was examined in a pybullet environment (see 5.3) we saw that the agent was straying slightly to the side, but was otherwise capable of walking with a satisfactory walk pattern (no excessive leaning, use of all legs, speedy locomotion,...). It should be noted that at this point the control bits $c_0 - c_3$ were set to [1,0,0,0] and we were only attempting to make the robot walk straight. As such we were only solving a relaxed problem to verify that our design was working correctly. There were problems, however, as closer inspection revealed.

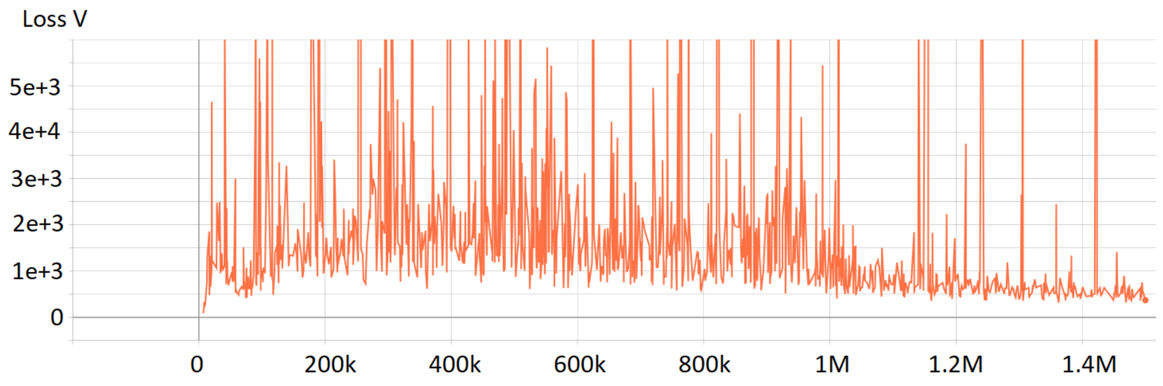
```
python agent_minitaur_inspired.py --mode=test --episodes=10 --loc=/runs/run_<X> --render
```

Listing 5.3: Testing the behaviour of the agent

When testing the agent's performance we noted that with the default episode termination scheme (episodes lasting from 8 - 15 seconds) the agent's walk cycle would after approximately 30 seconds completely fall apart and the agent would no longer be able to make any progress. We theorized that this was caused by the termination scheme not conveying the task we wanted the agent to perform. This was the main reason why we chose to develop the "death wall" scheme described in the previous section. The constant need to make progress rewards agents that can walk well with longer episodes, samples from which then make out a larger portion of the replay buffer. In this scheme badly performing agents get terminated only after a few seconds, while successful agents can act in the environment for up to 150 seconds (this length was chosen arbitrarily). After implanting this termination scheme we saw the problems of walk pattern falling apart disappear.



(a) Loss of Q1 network (the loss of Q2 network was similar so we omit it)



(b) Loss of V network

Figure 5.2: Losses of the first trained agent

Another problem was apparent when looking at the graphs depicting the V and Q loss function outputs over time. As can be seen in Figure 5.2, the losses that should have slowly decreased over time were behaving highly erratically. We addressed this issue in the subsequent training runs by decreasing the learning rate from the original 3e-4 to 5e-5 for both Q and V network. The V network was also implementing a target network with updates constrained using Polyak averaging method:

$$w_{new\_target} = (1 - p_{rate})w_{target} + p_{rate}w_{source}, \tag{5.1}$$

with the default p_{rate} set to 0.005 (which we decided to keep).

We attempted to extend this also to Q networks, but the Polyak averaging proved to deliver sub-par results, so instead, we replaced it with a simpler fixed-rate scheme. Using this scheme the source parameters would get copied to a target network only every N steps. In our case, we set this N to 1000. These changes did not need to be applied to the π - policy network as it was displaying good behaviour even without them Figure 5.3. Finally, we also reduced the size of the replay buffer from 1e6 to 5e5, to encourage the replacement of old samples with newer samples.

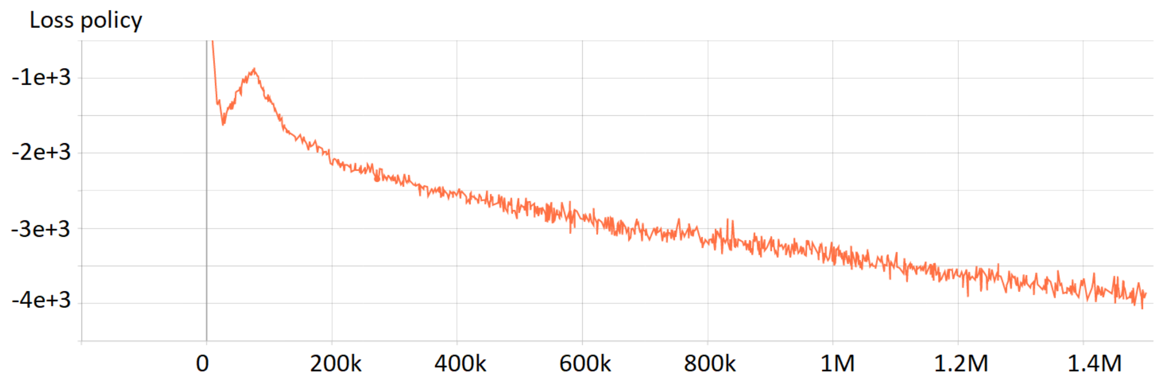


Figure 5.3: Policy network loss decreasing over time

Before running the training with these changes we reexamined the network architecture and decided to make one more change to it. We changed the architecture of the π network, by adding another, parallel branch between its input and output layers. This branch was significantly smaller than the already present 'main one', as it contained only 20 neurons per hidden layer. The idea behind adding this branch was that when training the π network in the simulation we would keep this 'side branch' frozen, exempting it from training. Later when we make the switch to a real-world environment we wouldn't have to train the entire network again and risk breaking it. Instead, we can freeze the main branch and unfreeze the side branch. This should in theory allow the network to adapt to the different parameters of the real world, shorten the training and help us avoid breaking the trained network. We test this hypothesis in the following section 5.3.

As the final attempt to make the learned policy more robust we also added a dropout layer (probability of dropout set to 0.1) after the input layer of the π network. By randomly zeroing some of the inputs it should help the network generalize better, which was displayed in [22].

The result of all these changes can be seen in Figure 5.4. The graph looks significantly different than the one shown in Figure 5.1 and this is mostly the result of the new episode termination scheme. Until the agent is able to learn a rudimentary walk pattern, episodes get terminated quickly not giving it enough time to accumulate any reward. Once the agent can walk (episodes 900+) it can get away from the death wall, quickly enough to start accumulating large rewards. After episode 1000 we see steady improvements in obtained reward as the agent learns to improve its efficiency.

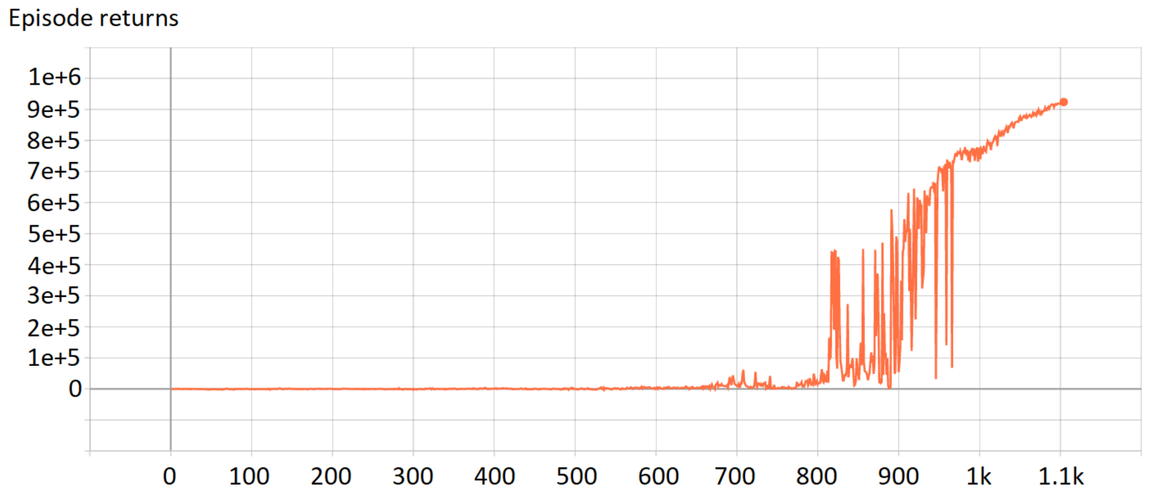


Figure 5.4: Reward collected per episode, modified network, robot walking straight only

The graph Figure 5.4 makes it obvious that rewards still seemed to be growing when the training ended. To confirm this we ran the training again, this time on 4 million frames. The sliding 100 episode average reward can be seen in Figure 5.5. Even after 4 million frames, there seems to be room for improvement, but the agent's behaviour was already satisfactory at this point, so further training was not conducted. Just as before we were only training the agent to walk forward.

In Figure 5.6 we see the losses of V (upper) and Q1 network (lower) obtained using the modified networks. We note that while spikes are still present the overall development of

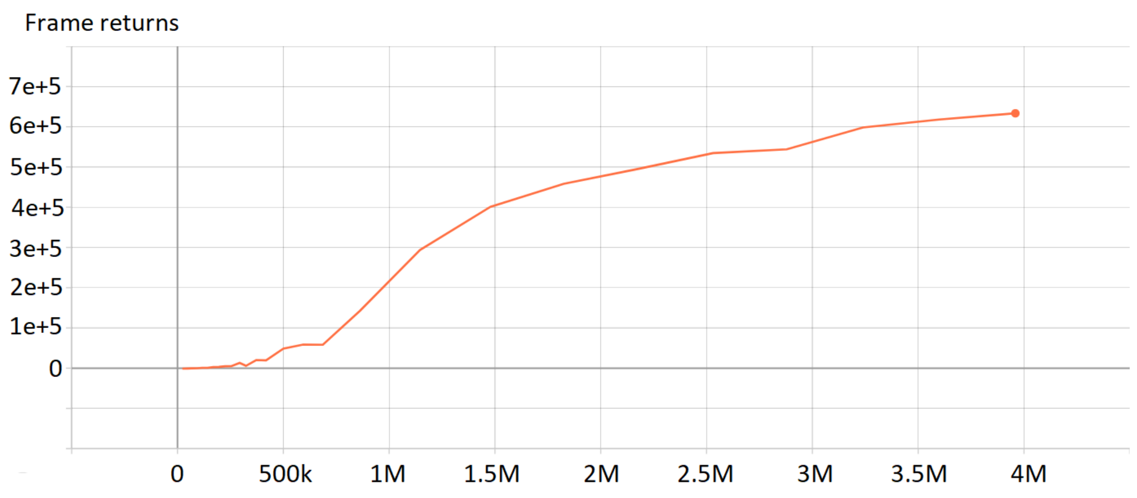
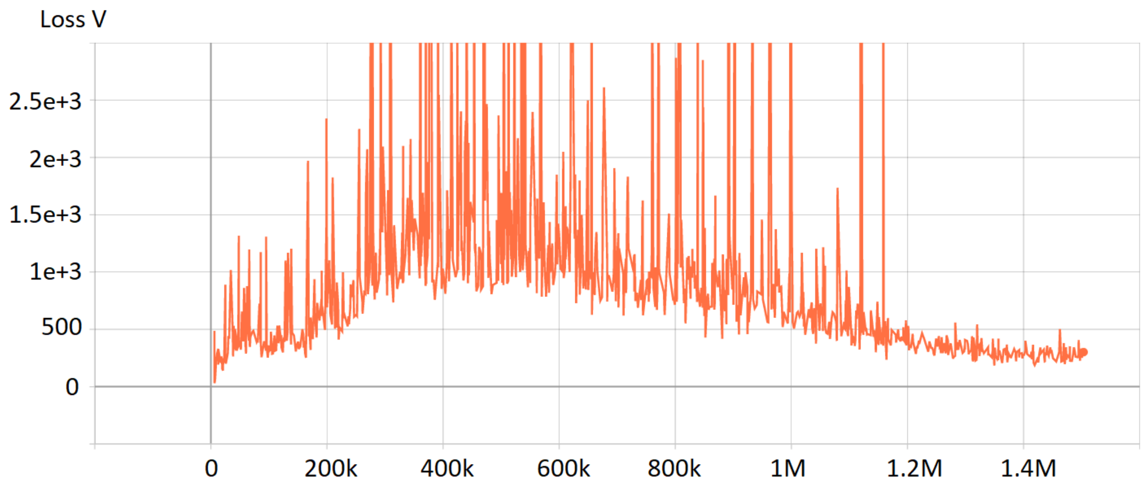


Figure 5.5: Average reward from 100 episodes, 4 million frames

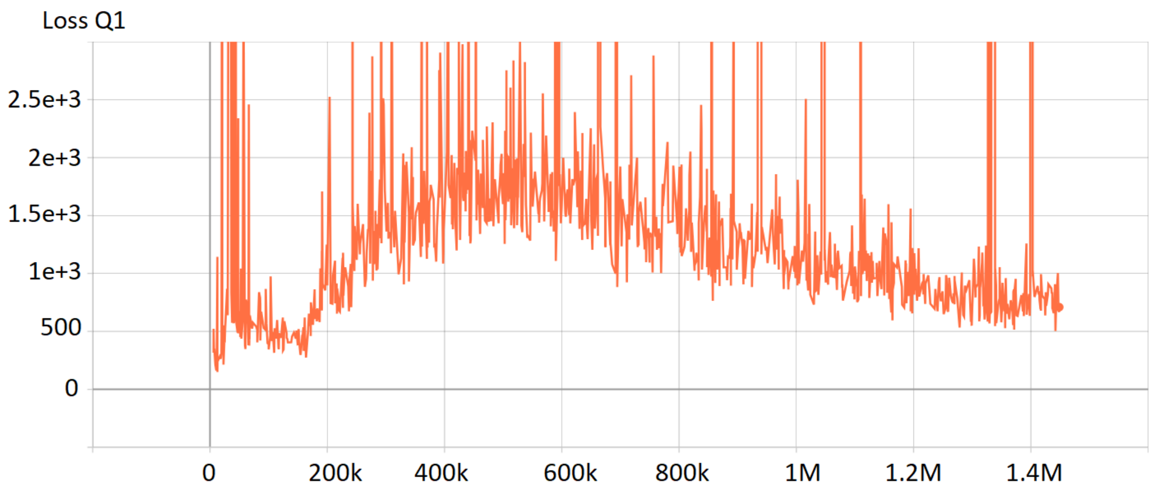
the losses seems to be much closer to the expected ideal. Both losses decrease over time signalling improved performance. The remaining spikes are possibly the result of training performed over batches of data. This means that a randomly sampled batch might contain data that displays 'bad' behaviour. This translate to a large loss that pushes the agent away from repeating this behaviour. It is possible that this problem could be further alleviated by using a larger batch size (the examples shown were obtained using a batch size of 100).

Because of all the modifications we were unable to reuse the previously obtained reward parameters and had to once again run a batch of random training sessions to find new parameters that worked. These newly obtained parameters were $(w_0, w_1, w_2, w_3) = (10833.473, 0.81, 65.764, 18.75)$. The value k_p of P regulators was still constrained in the range of $0.06 - 0.08$ (linearly increased with training frames to prevent the agent from making sudden moves at the start of the training).

However, even with this improved scheme we still observed some problems. One of them was that the agent did not fully utilize all of its limbs. The central leg on the robot's right side was permanently lifted and did not contribute to the robot's movement. This was to be expected since when it comes to walking on a perfectly flat plane six legs are not strictly necessary. With a degree of redundancy present, the agent was most likely not punished in any way by not using the leg and could save some energy by ignoring it. The situation would be different if the agent was expected to walk on an uneven plane. In that case, using all legs



(a) Loss of V network



(b) Loss of Q1 network

Figure 5.6: Losses of Q1 and V networks after modifications

would most likely lead to a more stable walk pattern and thus higher obtained rewards.

The second problem was more pressing. When examining the robot's trajectory Figure 5.7, we can clearly see that it is drifting from the X-axis by up to 10 meters. After examining the test data, we noted that the average step reward terms were:

- X-axis forward reward : 161
- energy expenditure punishment : -0.1
- Y-axis drift punishment : -0.1
- non-zero roll and pitch punishment : -1.4

We immediately note that the step reward was completely dominated by the forward

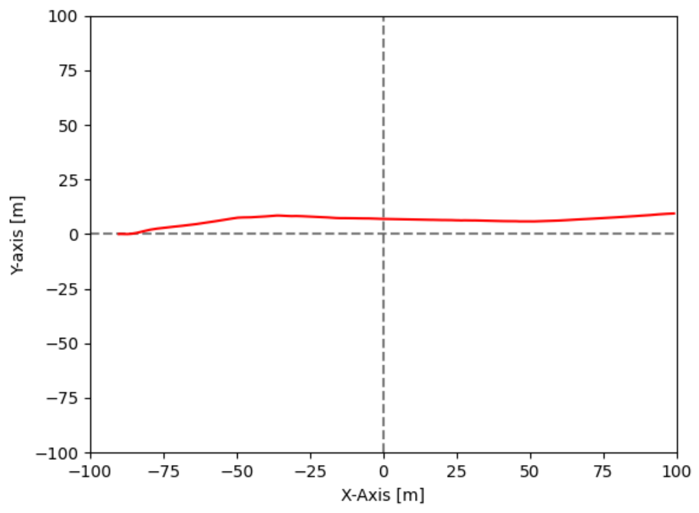


Figure 5.7: Robot's original trajectory of movement

reward term, with little contributions of other terms. During the training process, this was most likely not the case, but it still shows that the punishment terms could be scaled higher utilizing the $[w_1, w_2, w_3]$ parameters. If the punishment terms played a more significant role the robot's performance might be still improved.

Instead of retraining the agent from scratch and risk breaking it, we decided to run another 4 million frames of training on the already trained model. We changed the parameters $w_0 - w_3$ once again this time not relying on randomness, but changing the reward terms to $[w_0, w_1, w_2, w_3] = [6000, 100, 3000, 150]$, to make punishment terms more significant.

The resulting agent, while it still wasn't utilizing all robot's legs, displayed an even more stable walk pattern and significantly lower drift in the Y-axis Figure 5.8. Finally, we note that when we attempted to use the reward terms above for training an agent from scratch, the training failed. This goes to show the benefits of multiple-stage training when we build on top of an already well-performing agent in order to further improve its performance.

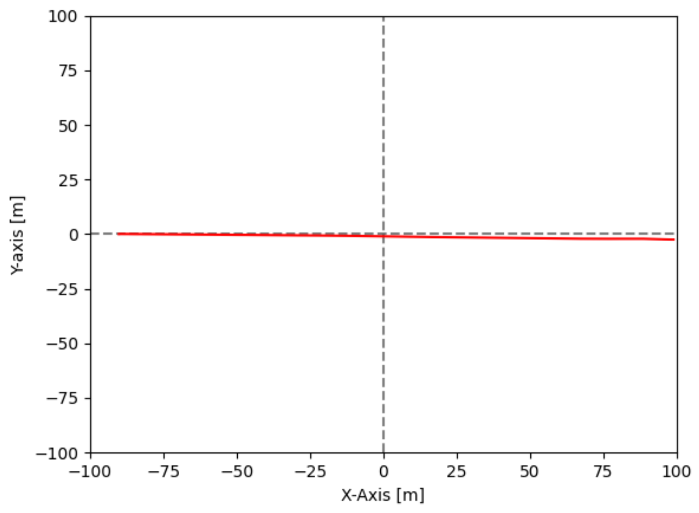


Figure 5.8: Robot's trajectory of movement after scaling the reward terms

5.2 Controlling the robot

In all previous examples, we simplified the task of controlling the hexapod robot. As such we only required the agent to learn how to walk straight, keeping the direction control bits $[c_0, c_1, c_2, c_3] = [1, 0, 0, 0]$. With the architecture of neural networks now confirmed to be working, we can address the problem of agent control.

We used the learned network weights obtained from the previous training as a starting point for the new training. The reward function was modified to take into account the requested direction when calculating the first two reward terms. During each step of the training, we programmed in a 1% chance of direction change, to simulate the user's direction change requests. With these modifications we ran the training again, using the reward parameters mentioned in the previous section.

As Figure 5.9 shows this approach was not successful. The agent failed to learn to change the direction of its movement according to the change in $c_0 - c_3$. We attempted to repeat the training with a significantly increased number of training frames. However, this attempt also failed. Recognizing that the used reward parameters and network weights might not be suited for the new task, we reverted back to training from scratch. As described in the previous section we ran multiple training sessions with randomized reward parameters. Still, the agent

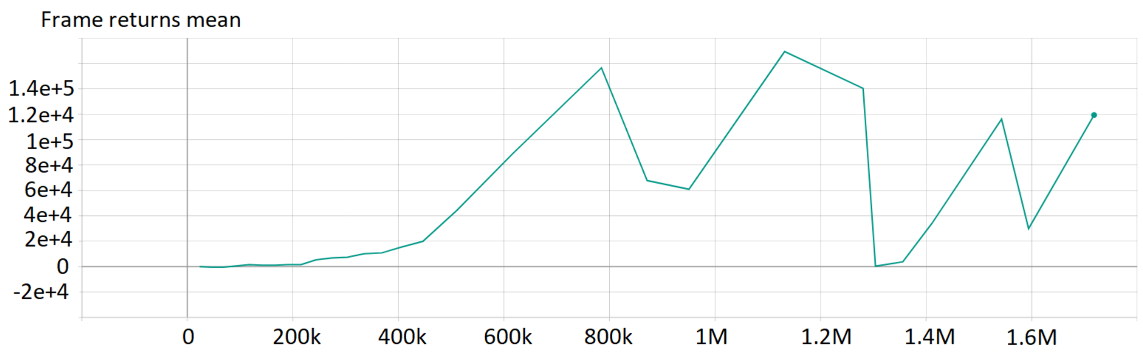


Figure 5.9: Agent failed to learn how to interpret the control bits

failed to learn anything. Lowering the frequency of direction changes also did not improve the agent's performance.

After these failures, we decided to abandon the idea of using control bits to change the agent's direction. While it is possible that with some changes to the network architectures or changes to the reward function this scheme could be made to work, we decided against pursuing this option. Instead, we resort to a cruder way of controlling the agent, which is the manipulation of its observed position in space.

Reverting to the trained model that learned to walk straight, we noted that the observation included the yaw angle of the robot's body. From this, we assumed that during the training the robot learned to correct its movement in the environment to keep its yaw close to 0 (parallel to the X-axis). If this turned out to be the case we could 'fool' the agent into thinking it has strayed off course by modifying the yaw angle. This in turn would prompt the agent to start turning.

We verified this by making the yaw angle dependent on the user's input and running a test episode while varying the yaw. It turned out we were correct and that the robot did respond to the changes in yaw. It could correct its course as long as the yaw angle was kept constrained to the range of $< -0.3; 0.3 >$ rad. Using this method we were able to exercise at least some degree of control over the robot.

5.3 Evaluating the parallel architecture

In the previous section, we introduced the idea of a frozen parallel branch of the neural network. We stated that by freezing this part of the network during the training in simulation and subsequently only training this side branch during further training in the real world, we could significantly speed up the second training process. To comprehensively test whether this assumption holds we would need to use the real nextro robot. However, since that isn't yet possible we resort to the use of simulation.

Starting with the pre-trained network obtained in the previous section, we slightly modify the behaviour of the simulation. The simulation parameters that are easily available to us are PID constants k_p, k_i, k_d of each motor and gravity acceleration g. The original and modified constants are listed below:

Original parameters	Modified parameters
• g = 10	• g = 11
• $k_p, k_i, k_d = 0.08, 0, 0$	• $k_p, k_i, k_d = 0.12, 0.01, 0.001$

These modifications weren't designed to necessarily simulate the real environment better, only to change the environment. This change simulates the changes that the agent would have to re-lean when transitioning from simulation into the real world.

With these modifications, we ran 10 test episodes with the pre-trained network and logged its performance. This showed that with the above changes to the simulation environment the agent could no longer control the robot effectively and so more training was needed. This was a good result as it allowed us to simulate the training process in an environment that the agent has not previously encountered.

We ran two more training sessions, each with identical reward parameters ($[w_0, w_1, w_2, w_3] = [6000, 100, 3000, 150]$) and the number of frames (1 million). Both trained agents were based on the original pre-trained agent from the previous section, so we were not training the agents from scratch. One agent's policy network had its side

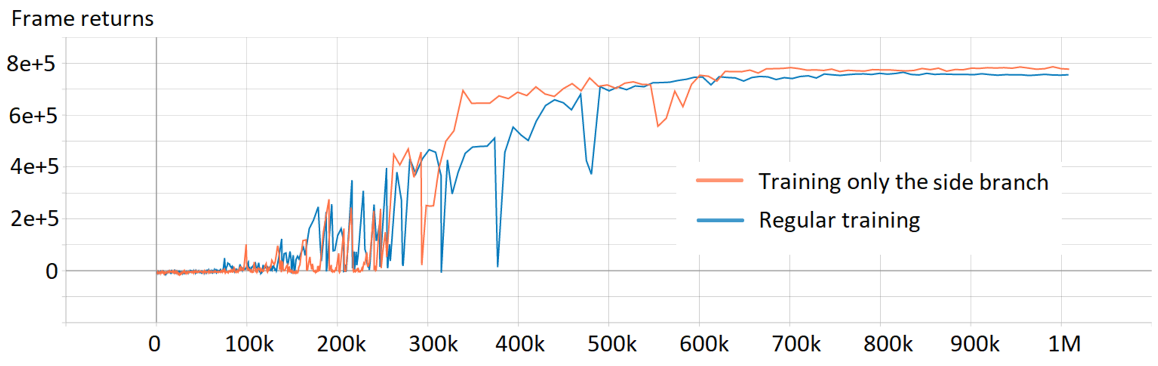


Figure 5.10: Comparison of training with unmodified π and side branch training reward

branch values frozen and only its main branch was being trained (same as all previous training sessions). The other agent had its main policy branch values frozen and only the smaller side branch was being trained. In both cases, all other networks (Qs and V) were trained as usual.

The results of these training runs can be seen in the Figure 5.10 below. We note that in the case of π network with the frozen main branch the training converged to high frame returns faster. It also appeared that training only the smaller side branch produced more stable learning. It is obvious that these properties, if they could be consistently replicated, would prove beneficial in the real world training scenario. However, we have to note that the overall performance in both training scenarios was fairly similar, with both agents producing stable returns after approximately 600 thousand frames of training. With the rate of processing at 30 frames per second this would translate into real training time of five and a half hours in both scenarios.

6 | Transitioning into the real world

As we have already seen when we tested the robot and learning algorithm in a simulated environment, we were able to achieve relatively good results. These results were at the time only contained to a simulated environment. But the simulated environment will always represent only an approximation of true reality. How the robot would perform in the real world is another question. While we will not answer that question in this thesis, we felt it would be wrong to ignore the topic altogether. So in this chapter, we present a detailed description of one possible approach of coupling the neural network to hardware. To enable this transition we will use a well-known collection of libraries and tools collectively known as ROS.

6.1 Robot Operating System (ROS)

When constructing any not-trivial robotic system we are bound to end up with a plethora of systems and subsystems. Managing the flow of this data generated by them can easily become a difficult task on its own. Without a robust method of tackling this problem, we are bound to end up with a tightly interconnected communications network. This creates yet more problems, as these systems tend to fall apart the moment any changes need to be made.

This is where ROS comes in. It would take too much time to present all of the ways that ROS can be used, but the official documentation [23] states that ROS is:

"an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. "

In this part of the thesis, we will focus on how we can use ROS to create a modular and, for the most part, hardware-independent way of connecting the RL network to the real world. Our goal will be to create an interface that can be easily modified and scaled. That way it could also be used for other types of robots and with different learning algorithms.

6.1.1 Communication within ROS

Let's start by examining how the communication works within ROS. ROS implements what is known as a publisher-subscriber model of communication. The way this works in practice is that the user creates nodes that represent logical or functional parts of the robot. Each of these nodes is capable of publishing *messages* (data) to *topics*. These topics can be *subscribed* to by a different node that can then access the messages. These nodes are grouped in packages that represent yet another layer in the hierarchy of the robotic system. To best way to illustrate this structure is with a simple example.

Imagine we have two packages one called *control_sys* and another called motor_sys. The nodes these packages contain can be seen below:

1. control_sys
   - control_node
2. motor_sys
   - motor_control
   - motor_feedback

In this arrangement, the motor_feedback node handles obtaining feedback from a motor using device-specific means (i.e. via encoder). It then publishes this data into a *topic* called for example *motor_RPM*. Control_node can then reach into motor_sys/motor_RPM and subscribe to all new messages that appear. Once the data gets processed in the control_node a new requested RPM can be pushed into the motor_sys/requested_RPM. Finally, the node motor_control subscribed to the requested_RPM topic the request message and through device-specific means communicates the request for RPM change to the motor.

This example shows the power of ROS. An engineer can encapsulate all the necessary functionality into packages/nodes and communicate information through topics/messages. In the example above if a new motor were to be used only the code in motor_control would need to be changed. Breaking down functionality into packages also opens the door for community-made and maintained packages. Already there are thousands of these available at https://index.ros.org/packages/.

ROS also makes network communication possible, by allowing nodes to connect even if they are not running on the same computer. This method might seem appealing for our use at first, since training the neural network requires computational resources that can't easily be put on a mobile platform. We could get around this issue by running the node/s responsible for the neural network on a powerful desktop computer while running the remaining nodes on a smaller computer with network capabilities on the mobile platform. While this approach is for sure feasible it also dictates the use of a Linux-enabled, networked computer on the mobile base of the robot. For our use, the tasks that this computer would be responsible for are fairly trivial as they for the most part consist of reading sensors and generating commands for the servos. Yet, this can also be achieved using a much simpler and cheaper micro-controller.

Taking this into consideration we instead decided to design the system, in a way that contains all nodes to a single desktop PC. In this arrangement, information can be forwarded to the onboard micro-controller from individual nodes using simple communication protocols such as UART or similar. When implemented this way we profit from the robustness of ROS communication and modularity. At the same time, we also get to keep the robot construction simple. In case it later became clear that a micro-controller is not enough, the robot-related nodes could easily be modified and launched from a more powerful onboard PC.

6.1.2 Designing the packages and communication

Because we have decided not to implement the learning algorithm on a real robot, the following section dealing with the ROS communication presents only a high-level overview. It presents one possible (but not necessarily the best) approach of implementing the interface between the robot and the RL agent. We avoided the use of more advanced features provided by ROS, such as services and parameter servers, and only utilize the publisher-subscriber in its most basic form. This is accomplished using nodes, topics and messages. An overview of this communication structure can be seen in the Figure B.1 in the appendix.

The nodes *nGymStepOutput* and *nGymStateInput* are contained in the gym environment and as such are responsible for fetching data for the agent (information about the robot's tilt, motor position, torque, etc.) and for publishing the information about the next requested joint angles. The nodes outside of the gym environment *nMotorPosCommand* and *nRobotState-Query* are responsible for directly communicating with the robot over the UART protocol. A single communication round is then:

1. Step method produces command for the joint motors (at the start of each episode reset method can also send a request for all zero angles).
2. Node nGymStepOutput publishes the request to the topic tMotorPosRequest.
3. Node nMotorPostCommand was subscribed to the tMotorPosRequest and fetches the new data.
4. The fetched data is relayed to the Nextro through the UART.
5. Nextro sets the angles and obtains data from its sensors.
6. Data is sent through the UART and is read inside the node nRobotStateQuery.
7. The data is published to the topic tRobotState.
8. In the gym environment the state method and its node nGymStateInput subscribed to the tRobotState reads the new data and uses it to calculate the new state and reward.

6.2 Hardware design and requirements

This section is concerned with the requirements that the robot and experimental environment have to fulfil to make learning in the real world possible. We will first discuss what data the robot needs to be able to supply and which sensors are therefore required. Afterwards, we will shift our attention to the experimental setup that will allow us to train the robot.

6.2.1 Nextro design

As was shown in Figure 3.1, Nextro is a robot actuated by 18 modelling servomotors. These motors, while being simple to control and use also have some disadvantages. Namely, without an external encoder, they cannot supply precise information about their current position. This can be resolved in two ways. One is rather straightforward and consists of adding a positional absolute encoder to each joint. These encoders can then be polled in fixed time intervals to obtain precise information about the joint's position. They also make it easy to compute velocity and acceleration should it be necessary. But this solution is impractical because it would no doubt prove difficult to attach these encoders to a robot that was not designed to accommodate them.

The second approach we propose is to presume that all joints can reach their target position in a very short amount of time. This should be the case as long as the difference between the original and target position $\Delta\beta$ is very small. In this case, we can then assume that the position of the joint is whatever the commanded position was. The criteria for $\Delta\beta$ is also easy to fulfil if we use a PID controller with a small P constant (in the simulation the value of 0.08 was used).

The other value each joint motor must supply is the applied force i.e. torque τ. A crude but in this case sufficient way of obtaining it is to connect the supply line of each motor through a current probe. Since all motors are otherwise identical and τ is proportional to current, we can use the current (multiplied by an arbitrarily chosen constant) as the proportional measure of the motor's torque.

Finally, we need to include an angle sensor, which will provide the yaw, pitch and roll of the robot. There are many of these on the market and they are very easy to use. For greater accuracy of the obtained data, we can use a combined reading from the gyroscope and accelerometer. The data from these can be combined using a simple complementary filter to obtain a more accurate measurement.

$$\psi_t = \kappa(\psi_{t-1} + \Delta\psi_{gyro_t}) + (1-\kappa)\psi_{acc_t} \tag{6.1}$$

, where $\Delta\psi_{gyro_t}$ is the difference $\psi_{gyro_t} - \psi_{gyro_{t-1}}$. The ψ_{acc_t} is obtained from the accelerometer by integrating and adding the obtained angle changes to it. As such it has a tendency to drift with time therefore it should be periodically reset to the value of ψ_t. κ determines the relative importance of accelerometric measurement and should be set to a low value ($\tilde{0}.99$).

These modifications make up the sensory equipment that needs to be added to the robot. Other measurements such as the robot's position will be addressed in the next section. The

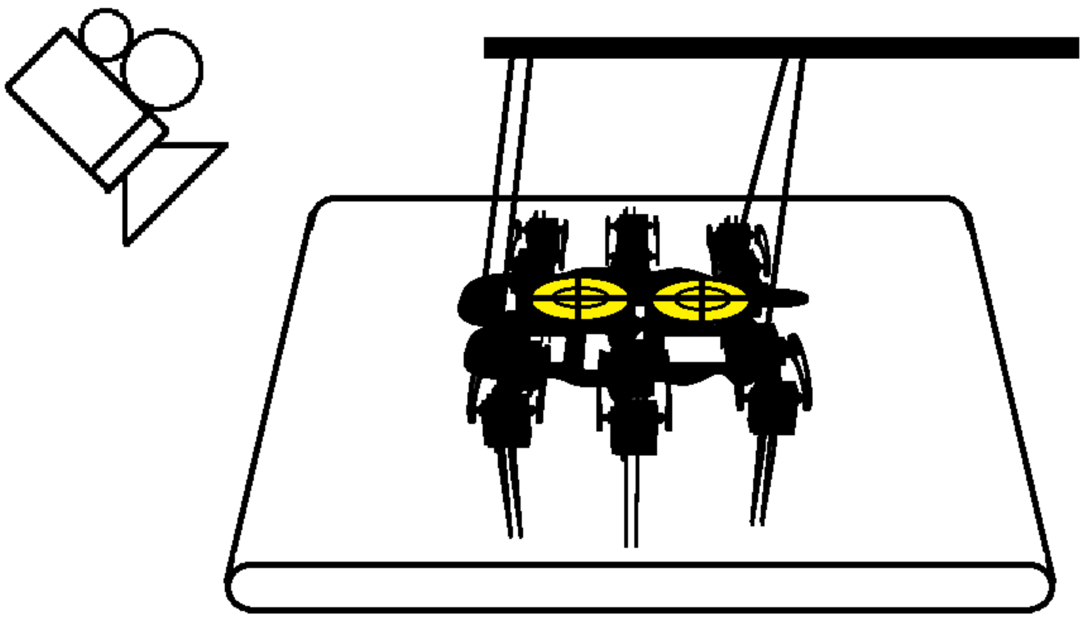


Figure 6.1: Example training setup

only other thing that needs to be decided is how to handle communications. Since we've already decided that we want to use a simple onboard computer that will not be running ROS, we are free to choose any communication protocol supported by both the onboard microcontroller and the main computer. Our options include but are not limited to UART, USB, Bluetooth, WiFi, ethernet,... etc. Of all these UART should be the easiest to implement, so we suggest using either that or WiFi if the selected onboard computer supports it. Since we will not be implementing the communication as part of this thesis, we will presume the use of UART.

6.2.2 Experiment design

When designing the real-world learning environment we need to keep in mind that the setup has to allow us to determine the position of the robot. This is crucial because it allows us to determine the reward terms which rely on it. Also since the robot will be in motion and could get damaged if allowed to roam unrestricted, we need to decrease the chance of it happening.

We propose a system with a loose harness and a conveyor belt, which determines the position of the robot using a simple camera and a set of tracking marks attached to the robot. The schematic view of this can be seen in Figure 6.1.

Using this scheme it is possible to adjust the speed of the conveyor belt to the robot's speed. The harness protects the robot from colliding with the conveyor belt or flipping over. Robot reset can be performed after every training episode by commanding the servos to a predetermined zero position. Done this way the training process should require next to no human intervention. Once the training is finished the robot can be steered by varying the input yaw angle as described in 5.2.

7 | Conclusion

The master's thesis dealt with the introduction of reinforcement learning methods and the possibilities of their application to control a hexapod robot in a simulated environment. The Soft Actor-Critic method was used in training with the result of the training process being an agent capable of controlling the robot. The whole process of implementing the training, including the design of the robot's model, environment, tasks and reward functions, has been presented. The trained agent was capable of ensuring a stable walk pattern in the forward direction. The thesis also includes a theoretical summary of the process that could be used to transition the solution from the simulated environment into the real world. Overall, the work has demonstrated the relevance of deep reinforcement learning in the design of walking robots and can serve as a basis for future works dealing with this topic.

Bibliography

[1] NIELSEN, M. *Deep Neural Network*. 2019. ADDRESS: ⟨http://neuralnetworksanddeeplearning.com/images/tikz36.png⟩.

[2] STUART J. RUSSELL, P. N. *Artificial Intelligence: A Modern Approach*. Pearson, 2009.

[3] LAPAN, M. *Deep Reinforcement Learning Hands-On*. Packt Publishing, 2018.

[4] GREGOR, M. *Towards Intelligent Agents using Deep Reinforcement Learning*. 2019. PhD thesis, University of Zilina.

[5] HAARNOJA, T. et al. *Learning to Walk via Deep Reinforcement Learning*. 2019. ADDRESS: ⟨https://arxiv.org/pdf/1812.11103.pdf⟩.

[6] HAARNOJA, T. et al. *Soft Actor-Critic Algorithms and Applications*. 2019. ADDRESS: ⟨https://arxiv.org/pdf/1812.05905.pdf⟩.

[7] HEESS, N. et al. *Emergence of Locomotion Behaviours in Rich Environments*. 2017. ADDRESS: ⟨https://arxiv.org/pdf/1707.02286.pdf⟩.

[8] SCHULMAN, J. et al. *HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION*. 2018. ADDRESS: ⟨https://arxiv.org/pdf/1506.02438.pdf⟩.

[9] YOON, C. *Vanilla Deep Q Networks*. 2019. ADDRESS: ⟨https://towardsdatascience.com/dqn-part-1-vanilla-deep-q-networks-6eb4a00febfb⟩.

[10] FUJIMOTO, S. – HOOF, H. van – MEGER, D. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. ADDRESS: ⟨https://arxiv.org/pdf/1802.09477.pdf⟩.

[11] YOON, C. *Double Deep Q Networks*. 2019. ADDRESS: ⟨https://towardsdatascience.com/double-deep-q-networks-905dd8325412⟩.

[12] OPENAI. *Vanilla Deep Q Networks*. 2018. ADDRESS: ⟨https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html⟩.

[13] OPENAI. *Proximal Policy Optimization*. 2018. ADDRESS: ⟨https://spinningup.openai.com/en/latest/algorithms/ppo.html⟩.

[14] SCHULMAN, J. et al. *Proximal Policy Optimization Algorithms*. 2017. ADDRESS: ⟨https://arxiv.org/abs/1707.06347⟩.

[15] OPENAI. *Soft Actor-Critic*. 2018. ADDRESS: ⟨https://spinningup.openai.com/en/latest/algorithms/sac.html⟩.

[16] HAARNOJA, T. et al. *Soft Actor-Critic:Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. ADDRESS: ⟨https://arxiv.org/pdf/1801.01290.pdf⟩.

[17] YOON, C. *In-depth review of Soft Actor-Critic*. 2019. ADDRESS: ⟨https://towardsdatascience.com/in-depth-review-of-soft-actor-critic-91448aba63d4⟩.

[18] REICHEL, S. – SZADKOWSKI, K. von – AL., et. *Phobos Wiki*. 2019. ADDRESS: ⟨https://github.com/dfki-ric/phobos/wiki⟩.

[19] ALLEVATO, A. *Create your own urdf file*. 2017. ADDRESS: ⟨http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file⟩.

[20] NOTA, C. – AL., et. *Phobos Wiki*. 2020. ADDRESS: ⟨https://github.com/cpnota/autonomous-learning-library⟩.

[21] LOSHCHILOV, I. – HUTTER, F. *SGDR: Stochastic Gradient Descent with Warm Restarts*. 2017. ADDRESS: ⟨https://arxiv.org/pdf/1608.03983v5.pdf⟩.

[22] SRIVASTAVA, N. et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. Vol. 15, no. 2, pp. 1929–1958. 2014.

[23] TEAM, R. *ROS Wiki*. 2020. ADDRESS: ⟨http://wiki.ros.org/⟩.

ČESTNÉ VYHLÁSENIE

Vyhlasujem, že som zadanú prácu vypracoval samostatne, pod odborným vedením vedúceho práce, ktorým bol doc. Ing. Michal Gregor, PhD. a používal som len literatúru uvedenú v práci.

Súhlasím so zverejnením práce a jej výsledkov.

Dátum odovzdania práce, Žilina

____

podpis

University of Žilina

Faculty of Electrical Engineering and Information Technology

28260220212001

REALIZING WALKING FOR A WALKING ROBOT USING DEEP REINFORCEMENT LEARNING

APPENDIX

2021

Bc., Daniel Adamkovič

List of appendices

Appendix A | Expected Grad-Log-Prob Lemma

Recalling that all probability distributions are normalized, we can write:

$$\int_x P_\theta(x) = 1 \tag{A.1}$$

Take gradient of both sides:

$$\nabla_\theta \int_x P_\theta(x) = \nabla_\theta 1 = 0 \tag{A.2}$$

We will need to use the log derivative trick which states that since:

$$\nabla_\theta \log f_\theta(x) = \frac{\nabla_\theta f_\theta(x)}{f_\theta(x)}, \tag{A.3}$$

then $\nabla_\theta f_\theta(x)$ can be written as:

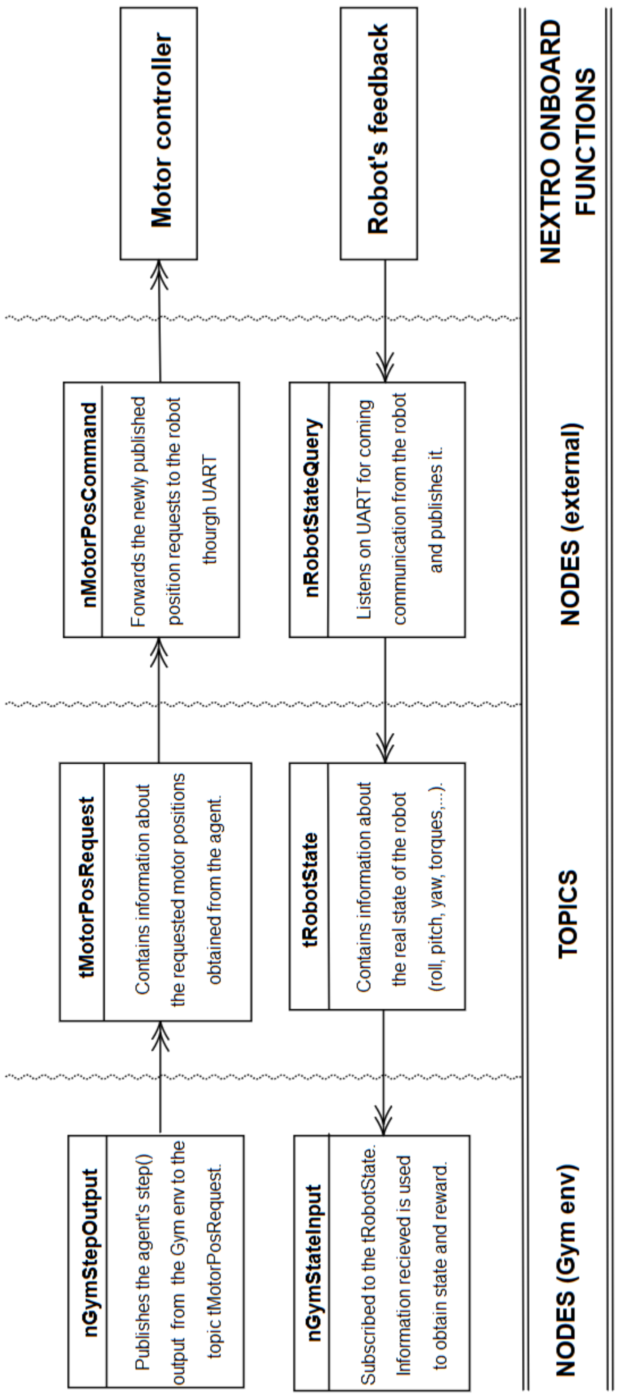
$$\nabla_\theta f_\theta(x) = f_\theta(x) \nabla_\theta \log f_\theta(x) \tag{A.4}$$

We can finally write:

$$\begin{aligned} 0 &= \nabla_\theta \int_x P_\theta(x) \\ &= \int_x \nabla_\theta P_\theta(x) \\ &= \int_x P_\theta(x) \nabla_\theta \log P_\theta(x) \end{aligned} \tag{A.5}$$

$$0 = E_{x \sim P_\theta}[\nabla_\theta \log P_\theta(x)]$$

Appendix B | ROS communications



nGymStepOutput
Publishes the agent's step() output from the Gym env to the topic tMotorPosRequest.
nGymStateInput
Subscribed to the tRobotState. Information recieved is used to obtain state and reward.
tMotorPosRequest
Contains information about the requested motor positions obtained from the agent.
tRobotState
Contains information about the real state of the robot (roll, pitch, yaw, torques,...).
nMotorPosCommand
Forwards the newly published position requests to the robot thourgh UART
nRobotStateQuery
Listens on UART for coming communication from the robot and publishes it.
Motor controller
Robot's feedback
NODES (Gym env)
TOPICS
NODES (external)
NEXTRO ONBOARD FUNCTIONS

Figure B.1: Overview of the ROS communication scheme

Appendix C | Resumé v slovenskom jazyku

Za posledné dekády sme boli svedkami mnohých objavov, ktoré by v minulosti boli považované za výplody fantázie autorov sci-fi. Náročnosť úloh, ktoré musia moderné roboty vykonávať stúpa a spolu s ňou aj nároky kladené na roboty. Tento nárast komplexity so sebou ale prináša aj problémy. Jedným z nich je to, že komplexné roboty vyžadujú komplexné riadenie, ktorého návrh je na pleciach technikov. Aj s využitím najmodernejších techník zostáva návrh riadenia chodiacich robotov netriviálnym problémom a úspech nie je nikdy zaručený.

V tejto práci je prezentovaný alternatívny spôsob riadenia robota s využitím metód hlbokého učenia s odmenou. Namiesto návrhu komplikovanej regulačnej sústavy sa vytvorí model robota a úlohy, ktoré má plniť. Následne sa z tohoto modelu získa informácia o tom, ako dobre alebo zle si robot počína a tá sa použije pri modifikácii jeho správania. Toto sa opakuje až do dosiahnutia želaného správania robota. Tento postup získavania spätnej väzby a modifikácie správania je podobný spôsobu, akým sa učia všetky živé tvory v prírode. Vďaka tomu, že tento postup je aplikovateľný na množstvo rozličných problémov, je v súčasnosti učenie s odmenou jednou z najrýchlejšie sa rozvíjajúcich metód v rámci celej umelej inteligencie.

Práca samotná pozostáva z týchto častí:

- **Teoretický úvod, stavové a strategické metódy**: Sekcia obsahuje úvod do problematiky učenia s odmenou a predstavenie základných pojmov, ktoré sú ďalej v práci používané. Primárne zameranie je na metódy založené na stratégii, pričom sú ale predstavené aj stavové metódy, ktoré sú neoddeliteľnou súčasťou stavových metód, a teda ich pochopenie je nutné pre prácu s nimi.
- **Postup pri návrhu prostredia, modelu robota a úlohy**: Samotné trénovanie agenta, ktorý ovláda robota, je možné aj v simulácii, pričom toto riešenie so sebou prináša

mnoho výhod oproti trénovaniu v reálnom svete. Použitie tohoto postupu je ale podmienené vytvorením modelu robota a navrhnutím úlohy, ktorú má agent splniť.

- **Zhrnutie výsledkov**: Evaluácia agenta bola uskutočnená po ukončení trénovania a pre dosiahnutie akceptovateľných výsledkov bolo potrebné vytvoriť viacero verzií tréningu. Tieto sa líšili rôznymi parametrami použitými pri tréningu a definíciou úlohy.
- **Prenesenie výsledkov do reálneho sveta**: Simulované výsledky poskytujú predstavu o očakávaniach, ktoré je možné mať od reálneho riešenia, v mnohých prípadoch je ale potrebné riešenie realizovať aj v reálnom svete. Praktické riešenie problému prenosu riešenia do reálneho robota, teda realizácia tréningu a programové riešenie, nie sú súčasťou tejto práce. Poskytujeme ale analýzu možnosti riešenia tohoto prenosu a navrhujeme spôsob, akým môže byť realizované.

C.1 Teoretický úvod, stavové a strategické metódy

Metódy strojového učenia možno rozdeliť podľa viacerých kritérií. Pre naše potreby je nezaujímavejšie rozdelenie na metódy:

1. **S učiteľom (supervised)** - Pre realizovanie učenia je potrebné zhromaždiť veľké množstvo dátových záznamov, ktoré predstavujú ukážku "ideálneho" správania. Úlohou agenta je tak v podstate naučiť sa funkciu, ktorá vstupy premapuje na požadované výstupy. Spätnú väzbu pre agenta predstavujú korektné výstupy.
2. **Bez učiteľa (unsupervised)** - Táto forma učenia slúži primárne na hľadanie vzorov v poskytnutých dátach a tieto typy metód teda zväčša nevyžadujú spätnú väzbu.
3. **Učenie s odmenou (reinforcement)** - Nie vždy je jednoduché zozbierať dáta o ideálnom správaní agenta. Táto trieda metód sa spolieha na to, že ako spätná väzba bude definovaná len tzv. odmeňovacia funkcia. Tá poskytuje informáciu o tom, ako sa darí agentovi plniť požadovanú úlohu. Pri snahe maximalizovať zozbieranú odmenu sa agent zároveň učí lepšie plniť úlohu, za ktorú je odmeňovaný.

Z rozdelenia vyššie jasne vyplýva, že pri návrhu agenta, ktorý dokáže úspešne riadiť chodiaceho robota, sa ako najperspektívnejšia možnosť javí použitie učenia s odmenou.

Aby bolo vôbec možné podrobnejšie opisovať akékoľvek metódy strojového učenia, je potrebné uviesť niekoľko definícií. Jedným zo základných pojmov je **agent**, ktorý bol už viackrát spomenutý, ale zatiaľ bez definície. Napriek tomu, že existuje viacero možných definícií tohto pojmu, pre naše účely postačí definícia, ktorú poskytli Russell a Norvig v [2] a ktorá môže byť parafrázovaná ako: *Agent je akákoľvek entita, ktorá niečo koná, pričom racionálny agent koná tak, aby dosiahol najlepší možný výsledok (alebo najlepší očakávaný výsledok v prípade prítomnosti prvku náhody).* V tejto práci sa pod pojmom agent vždy myslí racionálny agent.

Ďalším potrebným pojmom je **odmena (r)**. Veľmi jednoducho je odmenu možné definovať ako číslo, ktoré nesie informáciu o tom, ako dobre agent plní úlohu. V praxi môže byť frekvencia (nenulových) odmien nízka alebo vysoká a aj samotná odmena môže mať rôznu veľkosť v závislosti od typu úlohy, použitej metódy, atď. Odmenu agent zbiera pri prechode medzi jednotlivými stavmi.

Stav (s) zas opisuje všetko, čo sa nachádza na vstupe agenta. Stav by mal v ideálnom prípade spĺňať Markovovo kritérium. To tvrdí, že stav by mal obsahovať dostatok informácií, aby na ich základe bolo možné konať (aspoň teoreticky) ideálne rozhodnutia. Povedané inak, znalosť budúcnosti nie je závislá od minulosti, ak poznáme súčasnosť.

Na to, aby bol agent schopný cieľavedome skúmať stavový priestor (množinu všetkých možných stavov), musí byť schopný konať. Agent koná v každom okamihu cez **akciu (a)**, ktorá je zároveň jeho výstupom.

C.1.1 Metódy na báze hodnoty stavu

Z definícií v predchádzajúcej sekcii vyplýva, že agent sa pri svojej činnosti pohybuje medzi stavmi a zbiera odmeny za prechody medzi nimi. Je teda možné definovať **hodnotu stavu** $V(s)$, ktorá predstavuje očakávanú odmenu, ktorú môže agent z daného stavu zozbierať.

$$\begin{gathered} G = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ V(s) = E[G|s] \end{gathered} \tag{C.1}$$

Je možné definovať aj jemnejšie ohodnotenie, ktoré spája stav a akciu, ktorá bude v danom stave vykonaná. Vtedy hovoríme o hodnote akcie $Q(s, a)$.

$$Q(s, a) = E[r + \gamma V(s')], \tag{C.2}$$

kde E značí "expectation", teda očakávanie. V oboch definíciách hodnôt slúži γ (zvyčajne inicializovaná okolo 0.99) na zníženie hodnoty stavov, ktoré sa nachádzajú v budúcnosti a ich prínos je teda neistý.

Je zjavné, že nájdením Q alebo V funkcie vieme jednoducho ohodnotiť, nakoľko dobré je byť v danom stave. Rozhodnutia následne môže agent robiť pomocou stratégie π jednoduchým sledovaním stavov s najvyššími odmenami:

$$\pi(s) = arg \max_{a \sim A} Q(s, a), \tag{C.3}$$

Je nutné podotknúť, že takáto stratégia neberie do úvahy to, že prechody medzi stavmi nemusia byť deterministické. Primárne sme sa ale snažili predstaviť princíp stavových metód v jednoduchom prípade a bez zbytočných komplikácií.

V prípade, že stavy je náročné definovať alebo ak je ťažké jednotlivé stavy rozlíšiť (obraz, zvuk,..), je výhodné použiť na ich spracovanie umelú neurónovú sieť. Tá sa dokáže interne naučiť nielen reprezentovať jednotlivé stavy, ale jej výstup po natrénovaní môže byť zároveň výstupom Q alebo V funkcie.

S obľubou sa používajú napríklad DQN (deep Q network) siete. Ide o off-policy metódu, ktorá nepotrebuje pracovať len s dátami získanými aktuálnou stratégiou, ale využíva zásobník na ukladanie vzoriek zozbieraných z prostredia. Tieto sú neskôr náhodne vyberané a používané na trénovanie. Dôvody na takéto trénovanie, viac o tom, ako samotné trénovanie prebieha a ďalšie podrobnosti o metóde je možné nájsť v práci.

C.1.2 Metódy na báze stratégie

Použitie stavových metód nie je vždy možné, pričom v mnohých prípadoch môže byť hlavnou prekážkou to, že tieto metódy nie sú aplikovateľné v prípade spojitého akciového priestoru. Hlavnou myšlienkou týchto metód je tiež to, že po nájdení hodnoty stavov sme schopní presúvať sa medzi stavmi s vysokou hodnotou. Primárne nás ale hodnota stavu (alebo akcie) nezaujíma, používame ju len na to, aby sme mohli nájsť stratégiu π. Prečo teda nehľadať priamo π? Toto je hlavnou myšlienkou metód na báze stratégie.

My v práci využívame Soft Actor-Critic (SAC), ktorá spája výhody off-policy učenia a pracuje aj so spojitými výstupnými hodnotami. Keďže sa jedná o metódu vcelku náročnú na pochopenie a vysvetlenie, čitateľovi odporúčame oboznámiť sa s informáciami v kapitole 2.3.3.

Hlavnou myšlienkou SAC je, že agent by sa mal nielen snažiť maximalizovať odmenu, ktorú získa, ale mal by sa zároveň snažiť maximalizovať entropiu svojich akcií. Snaha o maximalizovanie entropie agenta núti byť si "menej istý" akciami, ktoré vykonáva, čo v konečnom dôsledku vedie k robustnejšej stratégii.

Vďaka tomu, že ide o off-policy metódu, je možné využiť zásobník, obsahujúci zozbierané dáta. To následne vedie k vyššej efektivite učenia, ktorá je výhodná hlavne v prípade, že tréning prebieha v reálnom svete.

C.2 Výber simulačného nástroja, implementácia modelu robota a návrh úlohy a odmeňovacej funkcie

Softvérovú realizáciu projektu sme sa rozhodli vytvoriť s použitím programovacieho jazyka python. Pre simuláciu prostredia sme zvolili populárny softvérový nástroj pybullet. Ten umožňuje importovanie modelu robotov vo forme URDF súboru, simuláciu fyziky a zobrazovanie robota. Nástroj je dostupný zdarma a obsahuje aj ukážky predpripravených simulačných prostredí.

Samotná úloha, ktorú robot plní, interakcie robota s prostredím, získavanie dát atď. je realizované cez rozhranie, ktoré poskytuje balíček gym. Ten je štandardne používaný v rôznych projektoch z oblasti AI, vďaka čomu ho podporuje veľké množstvo implementácií učiacich metód.

My sme sa rozhodli použiť implementáciu SAC z knižnice ALL, ktorá je samotná nadstavbou balíka nástrojov na návrh a interakciu s neurónovými sieťami pytorch. Keďže ALL poskytuje všetku funkcionalitu spojenú s SAC, mohli sme sa sústrediť na návrh architektúry a ladenie hyperparametrov. ALL túto práce zjednodušuje tým, že veľmi priamočiaro oddeľuje implementáciu učiacej metódy od architektúr sietí.

Model robota sme vytvorili v programe Blender, pričom pri návrhu sme využili rozšírenie Phobos. To umožňuje jednoduchú interakciu so štandardnými 3D modelmi vytvorenými v Blendri a ich rozšírenie o prvky, ako sú motory, kĺby, hmotnosti, zotrvačnosti, atď. Takto upravený model je následne možné exportovať do formátu URDF, ktorý je použiteľný v pybullete.

Návrh odmeňovacej funkcie a úlohy, ktorú má robot plniť, boli úzko prepojené. Rozhodli sme sa pre jednoduchú úlohu, ktorá spočívala v tom, že robot má prejsť čo najdlhšiu vzdialenosť v X-ovej osi. Navrhli sme viacero odmeňovacích funkcií, ktoré mali túto úlohu reprezentovať. Prvotná, veľmi naivná funkcia, odmeňovala robota odmenou rovnou celkovej prejdenej vzdialenosti v osi X. Takto formulovaná odmeňovacia funkcia, ale viedla k veľmi rýchlemu nárastu odmeny a nebrala do úvahy ďalšie nepriame požiadavky. Napríklad požiadavku aby robot vynakladal pri pohybe čo najmenšie úsilie a aby chôdza bola stabilná a neviedla k otrasom robota.

Po niekoľkých ďalších zmenách sme sa inšpirovali odmeňovacou funkciou implementovanou ako súčasť jednej pybullet ukážky. Finálna odmena robota tak závisí od vzdialenosti, ktorú robot urazil v X-ovej osi od posledného merania ΔX. Robot je ale trestaný za zmenu v Y-ovej osi ΔY, roll a pitch uhly (uhol natočenia voči osiam X a Y) a energiu vynaloženú na pohyb. Zmenou proporcií medzi týmito časťami celkovej odmeny je možné dosiahnuť stabilnú chôdzu.

C.3 Implementácia metódy učenia a zhodnotenie výsledkov

Pri návrhu architektúry neurónových sietí sme sa inšpirovali projektom Minitaur [5]. Vzhľadom na väčší počet kontrolovaných končatín a väčší stavový priestor sme ale rozšírili počet skrytých neurónov v každej z dvoch skrytých vrstiev na 255. Stav obsahuje informácie o súčasnej polohe kĺbov, ich rýchlosti, uhle natočenia tela robota a taktiež štyri kontrolné bity, ktorými sme plánovali ovládať smer pohybu robota. Stav taktiež obsahuje informácie o predchádzajúcej požadovanej polohe kĺbov (výstup stratégie). Všetky tieto informácie sú následne privádzané na vstup spolu s históriou N predchádzajúcich stavov. N bolo počas tréningu považované za hyperparameter, ale vo finálnej verzii siete sme použili $N = 5$.

Jednotlivé epizódy, počas ktorých boli zbierané dáta, mohli byť ukončené podľa viacerých kritérií. Epizódy končili, ak sa robot prevrátil, ak uplynul maximálny čas trvania epizódy, ak prešiel maximálnu vzdialenosť a v prípade, ak ho dobehla "stena smrti". Posledný koncept vyžaduje vysvetlenie.

Pri použití príliš krátkeho maximálneho času robot nemusí mať dostatok času na prejdenie celej vzdialenosti. Môže tak dôjsť k tomu, že robot sa naučí kráčať len po istú maximálnu dobu, čo sa neprejaví počas tréningu, ale až počas testovania. V prípade, ak sa zvolí príliš dlhý maximálny čas, epizódy, v ktorých sa robot nehýbe, zas trvajú pridlho. Prišli sme teda s koncept steny smrti, ktorá sa konštantnou rýchlosť blíži k robotovi. V prípade, ak sa robot hýbe rovnakou alebo väčšou rýchlosťou opačným smerom, dokáže prežiť dlhšie a zbierať odmenu. Zlé epizódy sú ale ukončené rýchlejšie a výsledky z nich sa neakumulujú v bufferi.

Pri tréningu boli hyperparametre volené náhodným výberom, až kým sa nevytvorila skupina natrénovaných agentov, ktoré boli následne vzájomne porovnané. Tieto porovnania a skúmanie dát získaných počas tréningu viedli k zmenám v architektúre neurónových sietí. Do siete aproximujúcej stratégiu bola pridaná paralelná vetva, ktorej účelom je urýchliť dotrénovanie robota v reálnom svete. Do oboch Q sietí bol zas pridaný model tréningu, pri ktorom dochádza k získavaniu výstupov z jednej siete a trénovaniu druhej, s následnými periodickými synchronizáciami. Viac o tomto koncepte je možné nájsť v práci.

Vo výsledku sa podarilo natrénovať agenta, ktorý je schopný riadiť robota tak, aby sa pohyboval napred. Pri pohybe robot do značnej miery nevyužíval jednu z končatín, čo bolo pravdepodobne následkom toho, že pri jednoduchom pohybe vpred na rovnej ploche bola táto končatina vyhodnotená ako nepotrebná a jej nevyužitím bolo možné ušetriť energiu. Nepodarilo sa ale natrénovať agenta tak, aby dokázal riadiť smer pohybu robota podľa kontrolných bitov. Nie je ale vylúčené, že inou definíciou úlohy a úpravou odmeňovacej funkcie by bolo možné dosiahnuť aj takéto riadenie.

Experimentovali sme taktiež s architektúrou π siete. Sieť pozostávala z dvoch paralelných vetiev, hlavnej s väčším počtom neurónov a vedľajšej s menším. Počas tréningu boli váhy vedľajšej vetvy zmrazené, teda váhy jej neurónov neboli počas tréningu menené. Po tom čo prebehol tréning v simulácií, boli podmienky v simulovanom prostredí zmenené a agent získaný z predošlého tréningu bol opäť trénovaný aby sa im prispôsobil. Tento postup simuloval prenesenie agenta zo simulácie do reálneho sveta, v ktorom sa robot správa odlišne oproti simulácií. Porovnávaný bol tréning dvoch identických agentov, pričom jeden z nich bol trénovaný so zmrazenou vedľajšou vetvou (ako pri prvom tréningu) a druhý mal zmrazenú hlavnú vetvu a trénovaná bola len tá vedľajšia.

Očakávali sme, že agent so zmrazenou hlavnou vetvou by mohol dosiahnuť dobré výsledky skôr, kvôli menšiemu množstvu trénovaných neurónov a natrénovanej hlavnej vetve. Z porovnania v Figure 5.10, ale vyplynulo, že rozdiely v rýchlosti tréningu boli relatívne malé. Napriek tomu sa javilo, že trénovanie len vedľajšej vetvy bolo stabilnejšie oproti klasickému tréningu hlavnej vetvy. Viac o potenciále tejto architektúry a bližší opis experimentu je možné nájsť v texte práce.

C.4 Návrh transferu riešenia do reálneho sveta

Navrhnutý spôsob učenia agenta principiálne funguje totožne aj v prípade trénovania v reálnom svete. Dodatočne je ale potrebné vyriešiť spôsob získavania informácií zo senzorov robota, fyzickú realizáciu frézovacieho prostredia a určiť, či trénovanie samotného agenta bude prebiehať v integrovanom počítači robota alebo na vzdialenom počítači.

Všetky tieto detaily adresujeme v záverečnej kapitole práce, v ktorej prezentujeme jedno možné riešenie tréningu v reálnom svete. Vzhľadom na to, že informácie o hardvérových úpravách robota sú obsiahnuté v práci, nebudeme sa nimi zaoberať aj tu. My sme sa rozhodli prezentovať riešenie, ktoré sa spolieha na trénovanie agenta na vzdialenom počítači, ktorý komunikuje s menej výkonným počítačom integrovaným v robotovi. Vďaka tomuto rozdeleniu nebude systém robota nadmerne zaťažovaný tréningom agenta a jeho úlohou bude iba riadiť jednotlivé motory a poskytovať informácie o svojom stave.

Pre vytvorenie flexibilnej a škálovateľnej komunikácie sme navrhli použite Robot Operating System ROS, ktorý pracuje na princípe publisher-subscriber. Navrhli sme, ako by mohla byť v praxi takáto komunikácia rozdelená tak, aby bol zabezpečený jednoduchý prístup k relevantným informáciám. Výhodou ROS je aj jeho veľká používateľská základňa, ktorá vytvorila balíčky umožňujúce jednoduchú integráciu nových modulov. Robot by tak mohol byť jednoducho rozšírený napríklad o kameru alebo reproduktor, čím by sa zvýšil počet jeho možných využití.

Na záver sme navrhli aj možné riešenie prostredia, v ktorom by mohol byť robot trénovaný. Šlo by o použitie pohyblivého pásu, na ktorom by bol robot zaistený prostredníctvom voľného postroja. Ten by zabránil prípadnému prevráteniu a poškodeniu robota. Pozícia robota by bola snímaná kamerou, ktorá by sledovala sériu značiek umiestených na tele robota. Pás by sa pohyboval proti pohybu robota, čím by sa značne zredukovala početnosť potrebných ľudských zásahov. Bez dotestovania tohoto systému v praxi nie je ale možné určiť, nakoľko by bol systém praktický, ani ako dlho by tréning v takomto režime trval. Tento výskum môže byť predmetom skúmania ďalších prác.

C.5 Záver

Diplomová práca sa zaoberala predstavením metód hlbokého učenia s odmenou a možností ich využitia pri riadení robota typu hexapod v simulovanom prostredí. Pri tréningu bola použitá metóda Soft Actor Critic, s pomocou ktorej bol úspešne natrénovaný agent riadiaci robota. Prezentovaný bol celý proces implementácie trénovania, vrátane návrhu modelu robota, prostredia, úlohy a použitej odmeňovacej funkcie. Výsledný agent bol schopný zabezpečiť stabilnú chôdzu robota v priamom smere. Práca taktiež obsahuje teoretické zhrnutie postupu, ktorým by bolo možné riešenie preniesť zo simulovaného prostredia do reálneho sveta. Práca preukázala relevantnosť učenia s odmenou pri návrhu chodiacich robotov a môže slúžiť ako základ pre ďalšie práce zaoberajúce sa touto tematikou.