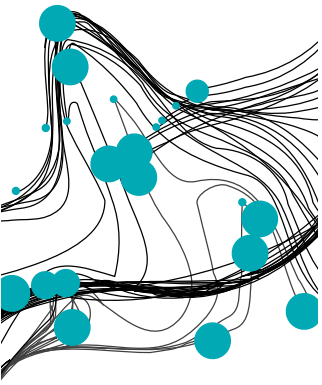
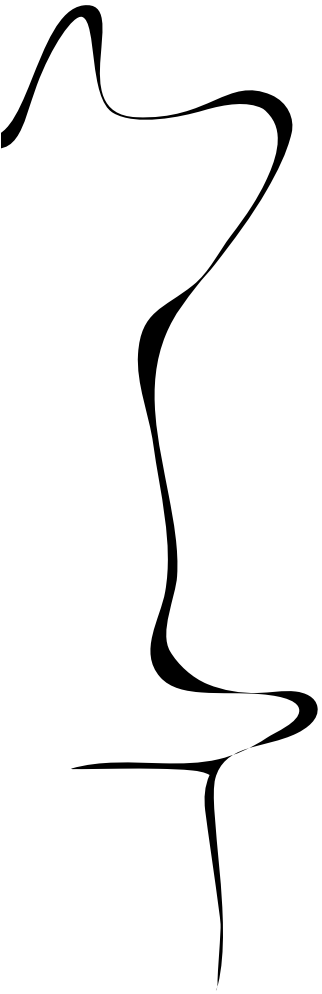


RAM ROBOTICS AND MECHATRONICS



Autonomous navigation of the PIRATE using Reinforcement Learning

L.J.L. (Luuk) Grefte

MSC ASSIGNMENT

Committee:
prof. dr. ir. G.J.M. Krijnen
N. Botteghi, MSc
dr. M. Poel

July 2020

023RaM2020
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Glossary

clamp rotate problem A main problem experienced in many experiments. Explained in Section 5.3.1.
clamp-reward Reward function to stimulate clamping.
clamp-drive policy Specific policy used in the HRL approach, for driving straight pipes. Elaborated in Section 3.3.1.

depth-reward Reward function to stimulate the camera to look through the pipe.

enter-turn policy Specific policy used in the HRL approach, for entering a turn. Elaborated in Section 3.3.1.

main-reward Commonly used reward function within the thesis, with the goal to move the PIRATE forward. This is explained in Section 3.2.4.
master policy Specific policy used in the HRL approach, which manages the other policies.
medium observations Observations group with the most observations, except the visual part is left out. Explained in Chapter 3.
minimal observations Observations group with the least amount of observations. Explained in Chapter 3.

outward-turn policy Specific policy used in the HRL approach, for leaving a turn. Elaborated in Section 3.3.1.

PyRep PyRep is a toolkit for robot learning research.

Ray Ray is a Reinforcement Learning framework.
RLlib RLlib is a Reinforcement Learning library part of the Ray framework.

stretch-reward Reward function to stimulate stretching.

TensorFlow TensorFlow is an open source platform for machine learning.
Tune Tune is a Reinforcement Learning tuning library part of the Ray framework.

vision observations Contains the medium observations, including visual depth observations. Explained in Chapter 3.
V-REP V-REP, currently called CoppelliaSim is a robot simulator.

Acronyms

ACER Sample Efficient Actor-Critic with Experience Replay.
ANN Artificial Neural Network.

CNN Convolutional neural network.

DDPG Deep Deterministic Policy Gradient.
DRL Deep Reinforcement Learning.

FuN FeUdal Network.

GAE Generalized Advantage Estimator.

HRL Hierarchical Reinforcement Learning.

LiDAR Light Detection and Ranging.
LSTM Long Short-Term Memory.

MC Monte Carlo.
MDP Markov Decision Process.

PAB Partially Autonomous Behaviours.
PBT Population Bases Training.
PID Proportional–Integral–Derivative.
PIRATE Pipe Inspection Robot for AuTonomous Exploration.
PPO Proximal Policy Optimization.

RaM Robotics and Mechatronics.
RL Reinforcement Learning.
RNN Recurrent Neural Network.
ROS Robotic Operating System.

SGA Stochastic Gradient Ascent.
SLAM Simultaneous Localization and Mapping.
SMDP Semi Markov Decision Process.
SRL Single Reinforcement Learning.

TD Temporal-Difference.
TRPO Trust Region Policy Optimization.

Summary

Pipe systems are the backbone of several important industries, which deal with gas, oil, water and sewage systems. As a leak in one of these pipes could be disastrous, it is important that these pipes are inspected regularly. These pipes often cannot be reached directly, as they are insulated or underground. There are several proposed solutions such as camera systems which can be deployed to inspect the pipe. However, these solutions often cannot deal with bends, long distances, insulation and complex control mechanisms. Autonomous pipe inspection robots could overcome these problems.

The PIRATE is a Pipe Inspection Robot for AuTonomous Exploration developed at the research group Robotics and Mechatronics (RaM) at the University of Twente (Dertien, 2014). The PIRATE is a worm like robot, which consist out of several modular sections. Because of the worm like design, the PIRATE is able to clamp in pipes with different diameters. Furthermore, it has rotatable section in the middle such that it is capable of making sharp bends.

In this thesis Reinforcement Learning (RL) is used to research and design autonomous navigation of the PIRATE in the simulation environment V-REP. In earlier work, simpler models and environments have been researched. In this research the PIRATE and simulating environment are modeled more completely and realistic. In the simulation environment V-REP, a realistic and challenging 3D pipe system is constructed, consisting out of sharp 90° bends.

For the implementation of the RL framework, the open source library Ray is used. Ray can be highly customized, but is also very easy to use out-of-the-box. The RL algorithm used in this thesis is Proximal Policy Optimization (PPO).

Several experiments are performed, which can be divided into four groups, namely feasibility, hyperparameter searches, Single Reinforcement Learning (SRL) and Hierarchical Reinforcement Learning (HRL) experiments. To test the newly constructed environment and model of the PIRATE, the feasibility of the designs had to be tested. After the feasibility experiments succeeded, proper hyperparameters for the RL algorithm are searched in other experiments. The hyperparameters of the RL algorithm and environment highly influence the outcome of the solution. Therefore, they are carefully selected.

After these experiments, the so-called Single Reinforcement Learning (SRL) experiments are performed. In these experiments RL is used in combination with several observation groups which were tested for their effectiveness. After the results of the SRL experiments were performed, the PIRATE was not able to properly enter and leave a bend. To solve these problems in the SRL experiments, Hierarchical Reinforcement Learning (HRL) experiments are performed.

In a HRL implementation several policies can be used to solve the problem. In the HRL experiment three policies are used. One policy is intended for driving through straight pipes, one policy for entering turns, and one policy for leaving turns. The specialized policies allowed the PIRATE to make sharper turns and increased performance is achieved. There is however a significant disadvantage to HRL compared to SRL. The setup and training of HRL is much more complicated then the SRL.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Pipes are present in refineries, chemical plants, power plants, industrial ships, sewer, gas and water distribution networks. These extensive pipe systems need regular inspections to maintain reliability. There are ways to inspect the pipes from the outside, for example by using x-ray solutions (VJ TECHNOLOGIES, 2013). However, since most pipes are heavily insulated or underground, this is not a usable solution. Especially as some pipes are not even made from metal.

For example, common types of sewer pipes made of PVC have a design life expectancy of 10 decades. However, in most cases the application life expectancy is much smaller. The pipe might be exposed to several temperature differences (Parvez, 2018; Folkman et al., 2012), which drastically reduces the life span to about 60 years. As various old apartments contain sewer pipes over several decades old, this forms a problem. Particularly because the sewer pipes within those apartments are generally not accessible by hand. The leakage could originate in several places within a multi-unit building. To discover the leakage problem several walls might have to be broken out. This might have significant impact on the inhabitants. To make things even worse, there are other materials used for sewer drainage pipe systems with even higher failure rates. Materials used include concrete, cast iron and asbestos.

Solutions for checking damages in sewer systems use a camera which is injected in the sewer from above. Long and sturdy cables are attached to these cameras, such that these cameras can be pushed through and retrieved. An example of such a camera is illustrated in Fig. 1.1. A firm spring is attached to the front so it can go through corners. The disadvantages of these systems is that they are hard to control in a complex pipe system such as T-sections. Furthermore, they cannot easily go up and due to the limited cable length, long distances are not possible.

Another example that illustrates the necessity of pipe inspections, would be the pipe systems on industrial ships. These pipes have an extraordinarily hard life. These pipes are exposed by salt water from the outside and by corrosive fluids from the inside. They also might have to operate under very high and low temperatures. Damage to a pipe can have serious consequences, for example pollution and fire hazards (Murdoch, 2012).

Due to the severity of the problems which could arise due to possible pipe damages, entire pipes will be replaced if there is the slightest of doubt in terms of the quality. This not only could halt production completely, but enormously increases the maintenance cost. In circumstances like these, it can be very valuable to have exact information about the current state and location of a potential problem to a pipe.

To solve these problems several pipe robots are developed to perform an inspection from the inside. Several of these robots are discussed in Section 1.3. Nowadays, these robots are be-

Figure 1.1: Commonly used sewer pipe inspection device, Testrix TX-120 (test-equipment.com, 2019).

coming increasingly autonomous. At the research group Robotics and Mechatronics (RaM) the Pipe Inspection Robot for AuTonomous Exploration (PIRATE) is being developed. The PIRATE is shown in Fig. 1.2. The PIRATE is a worm like robot, with a rotating section in the middle. Worm like robots are adoptable for propagation inside pipes with different diameters. However, worm robots will face difficulties if they have to move through bends. Controlling the PIRATE by hand is extremely difficult, therefore making this robot more autonomous is beneficial. Especially if complex maneuvers through corners are required. This is because the PIRATE consists of multiple joints that need to be controlled simultaneously. Furthermore, a human operator is not able to see the PIRATE from the outside. The schematic representation of the PIRATE can be seen in Fig. 1.2

The control of the PIRATE could be simplified by defining higher level functions such that independent joints and wheels can be controlled simultaneously. Due to the amount of wheels and joints, driving forward is not a straightforward task. If the PIRATE could be controlled by driving forward or backward, more autonomy is already achieved. In fact something like this is already attempted, this is treated in the Section 1.3. In order to simplify the control of the PIRATE defining higher level controllers could be the first step to consider. However, a complete different approach might be possible. Humans and animals can learn by interacting with the environment. When an infant plays and tries to interact with the environment it has no direct teacher. However, it does get feedback from the sensors of the human body. Exercising this relation can give a considerable amount of information about cause and effect of certain actions. Interactions with the environment play a huge role for humans and animals in learning new skills and obtaining knowledge. Reinforcement Learning (RL) is the computational approach to learn from interactions. This does not mean that RL is based on how humans and animals learn. RL simply tries to find effective learning methods (Sutton and Barto, 2018). This thesis researches if RL can be used to increase the autonomous behavior of PIRATE.

1.2 Problem Statement

The goal of this thesis is to control the Pipe Inspection Robot for AuTonomous Exploration (PIRATE) autonomously by Reinforcement Learning (RL) in the simulation environment V-REP. The pipe line system will be challenging such that the PIRATE learns a robust algorithm which should work within the specified simulation environment. To accomplish this, the PIRATE learns to make some difficult maneuvers such as clamping, turning and rotating. Furthermore, it is learning to time its actions correctly. The execution of a turn maneuver is shown in Fig. 1.3. Although this is a good representation of making a turn, by using RL, the agent could find better alternatives. Some of the real life obstacles, such as acquiring the exact position of the PIRATE within a 3D space, are out of scope of this research.

1.2.1 Research Question

Based on the problem statement the research question is formulated below. Beneath the main research question, several sub-questions are stated.

> What is required to fully autonomously drive the Pipe Inspection Robot for AuTonomous Exploration (PIRATE) within the simulation environment V-REP by Reinforcement Learning?
>
> - Which observation groups can be beneficial for the PIRATE?
> - Which reward functions can increase the performance of the PIRATE?
> - To what extent can an LSTM improve the performance of the PIRATE?
> - To what extend can a hierarchical action structure help in improving the performance of the PIRATE?

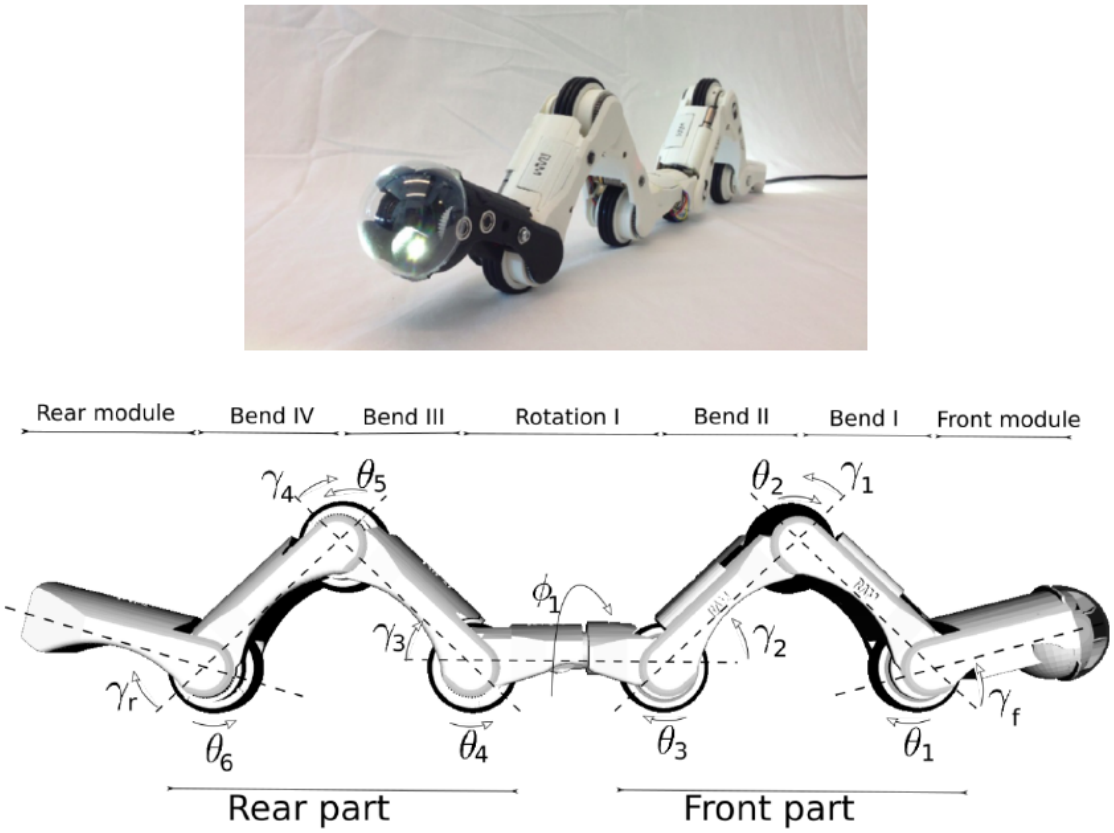


Figure 1.2: Upper: Real realization of the PIRATE. Lower: Schematic representation of the PIRATE, showing the joint (θ) and wheel (γ) orientations (Geerlings, 2018).

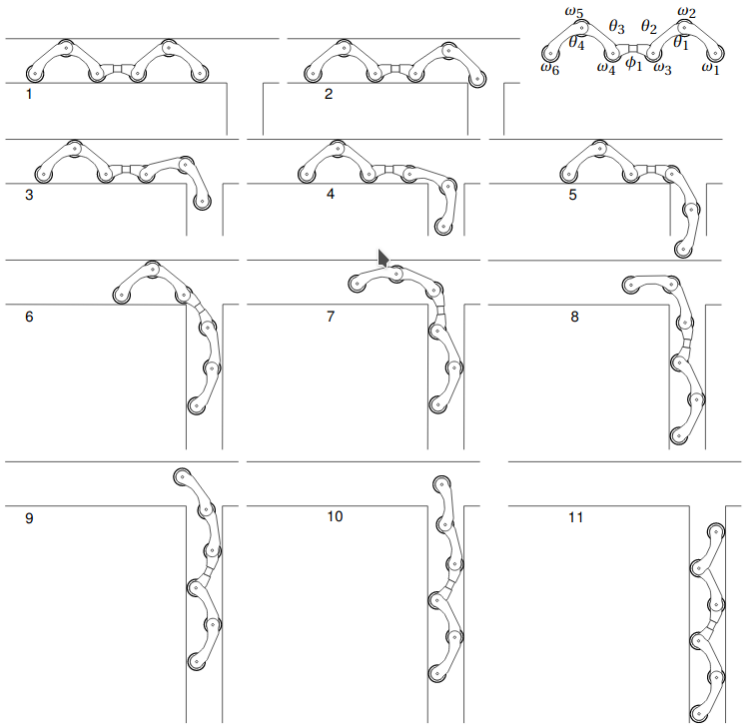


Figure 1.3: A schematic representation of the PIRATE taking a turn (Dertien, 2014).

1.3 Related Work

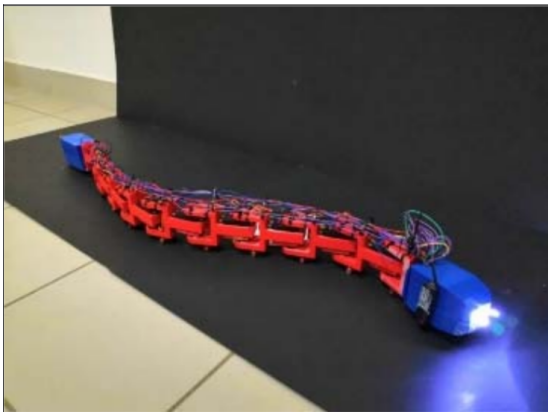
The Pipe Inspection Robot for AuTonomous Exploration (PIRATE) was introduced in 2014 by (Dertien, 2014), where several mechanical and control frameworks are proposed. Another software control framework to increase the autonomy of the PIRATE is proposed by (Garza Morales, 2016), where several Partially Autonomous Behaviours (PAB) are implemented. Both mechanisms work by controlling a MIDI control panel.

The first work on the PIRATE which involves RL, is based on autonomously navigating the PIRATE in a constrained pipe-like environment making a turn (Barbero, 2018). In that research a planar arm with 3 rotational joints is trained to reach a given target under the presence of pipe-like wall. The situation can be compared to the first trajectory of beginning a turn, illustrated in 4, 5, 6 in Fig. 1.3. A second implementation to autonomously navigate a turn is researched by Zeng (2019). The main difference with Barbero (2018), is the simulation environment and algorithms used. Zeng (2019) uses the RL algotrithm Proximal Policy Optimization (PPO), compared to Deep Q-learning in Barbero (2018). Both manage to make the turn with their robot in their simplified environment.

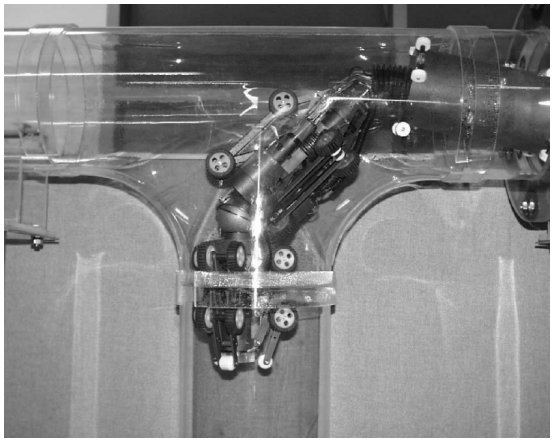
Outside of the University of Twente, several pipe inspection robots are also developed. A few of these are shown in Fig. 1.4. The time span of the robots shown is between 1999 and 2019, of course during this time development continued. MAKRO is first developed in 1999 (Rome et al., 1999). The goal for MAKRO is to function as an autonomous sewer robot. It has flexible joints which make it possible to go up, left, right and down. However, the MAKRO lacks a clamping mechanism, as it is driving through sewers and not clamping in them. In the years after, more work is done to increase the autonomy of the Makro (Rome et al., 1999). Another pipe robot which was developed in the early 2000s, is a robot coming from Sngkyunkwan University (Choi and Ryew, 2002). This pipe robot is intended for internal inspection of urban gas pipelines. It has a complex joint design, which allows it to lock its joints. This enables it to create a stiff section, which is convenient when it needs to cross a T-section. Also it is able to clamp itself in a pipe which makes vertical movement possible. Another interesting design is the Kantaro (Nassiraei et al., 2007), which is also intended for sewer pipe inspection. As can be seen in Fig. 1.4(e), the Kantaro is able to clamp itself in a pipe by stretching its handles to the side. Another sewer inspection pipe is presented in Abdellatif et al. (2018). The disadvantage of this pipe inspection robot is that it is static and is not able to clamp itself properly. However, the advantage is that it is a relative simple robot to build and control. A real snake like robot is presented in (Selvarajan et al., 2019). Huge flexibility is achieved, due to the amount of segments which can freely move. However, there is no way in which the robot can clamp vertically. Another interesting pipe robot is the PipeTron, which actually bares resemblance to the PIRATE robot. A major difference compared to the PIRATE is the rotating section. In the PipeTron rotation is achieved by a special yaw zig-zag configuration, which allows for rotation of the whole robot. For a more elaborate explanation, see (Debenest et al., 2014)

1.4 Outline

The remainder of this thesis is structured as follows. In Chapter 2 the background required for this thesis is stated. Chapter 2 treats Reinforcement Learning (RL), the PIRATE itself and some basic information about the tools used. After the background, in Chapter 3 the analyses and methodology are presented. In this chapter the problem is analyzed. Furthermore, the methods used for the upcoming experiment designs are considered. The Design and Implementation of these experiments are included in Chapter 4. The results of the methodology and the design are presented and discussed in Chapter 5. At the end, the conclusion is given in Chapter 6.



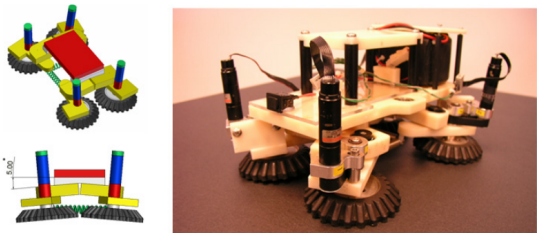
(a) Snake-Robot (Selvarajan et al., 2019).



(b) Inspection of urban gas robot (Choi and Ryew, 2002).

(c) Pipe inspection robot (Abdellatif et al., 2018).

(d) MAKRO robot (Adria et al., 2004).



(e) Kantaro (Nassiraei et al., 2007).

(f) PipeTron (Debenest et al., 2014).

Figure 1.4: several pipe inspection robots.

2 Background

In this chapter background information required for the thesis is supplied. The first three sections are about Reinforcement Learning (RL). In the first section the key parts of RL are treated. The second section will extend this to Deep Reinforcement Learning (DRL), where the third section will introduce Hierarchical Reinforcement Learning (HRL). The remaining sections introduce the PIRATE, and Ray.

2.1 Reinforcement Learning

RL is one of three main areas of machine learning, besides supervised and unsupervised learning. Supervised and unsupervised learning are briefly explained first. After that, a general explanation of RL is given, which will lead to a deeper discussion of RL.

Supervised learning trains an algorithm using labeled data. This could be for example images of cats and dogs. The goal is to train the algorithm such that it is capable of distinguishing between brand-new images of cats and dogs. When a labeled image is put through, the algorithm makes a prediction. Should the prediction be wrong, the algorithm adjusts itself slightly. After training, unlabeled images can be put through the algorithm to classify these images as either a cat or a dog.

Unsupervised learning attempts to find a pattern in unlabeled data. It groups data points with similar features together. If new data is added, it is able to appoint the data within a certain group. Therefore, it is also referred as a self-organizing algorithm.

In contrast to supervised and unsupervised learning, the data in RL is created by running an agent in an environment. The agent in the environment has to figure out by itself which actions it should take to maximize its reward. On the contrary to supervised learning, the learner is not precisely told which actions it should take. The agent has to use trail and error to learn the optimal behavior. A huge advantage of RL over supervised learning is the capability of creating data, for example by using a simulation environment.

In Section 2.1.1, some important parts of RL are introduced. The next sections specify more in depth information about RL. All the information in Section 2.1 is based on the lectures from (Isbell and Littman, 2016) and (Silver, 2015).

2.1.1 Main Elements of Reinforcement Learning

The basic elements required to define a RL problem are an agent, an environment and a reward structure. In this subsection several important elements of RL are introduced. The next subsections will elaborate on this.

Agent and Environment

Two important elements are, of course, the agent and the environment in which the agent operates. The agent can be seen as the controller in the environment. The agent receives observations and rewards and uses these to decide which actions to take. The environment encompasses everything except for the agent. The agent interacts with the environment as shown in Fig. 2.1.

Goal

The agent has a certain objective it wants to achieve. If for example, the environment is a game, winning the game is the goal of the agent. The reward function has a direct relation with the goal. If a game requires to have the most points to win, a suitable reward function could be receiving a positive reward if a point is obtained. Maximizing the reward is then directly related

to winning the game. Of course, different reward functions are possible. The agent might receive reward for stealing points of opponents or only for winning the game itself. How rewards are defined, could determine in what way the goal is achieved.

Reward Function

As already mentioned in the previous section, the reward function is directly related to the goal of the agent. The reward function maps each state-action pair with a specific scalar reward. Although the reward function can be rather complex and take into account several aspects, it will always be single scalar. The reward function gives the agent direct feedback whether an action was beneficial or not, and thus simply defines the goodness of a certain state-action pair. Due to the feedback the agent receives, the agent is able to discover the desired behavior in the environment.

Value Function

Something not mentioned yet, but equally important as the reward function is the value function. Where the reward function maps direct feedback, the value function defines how good it is in the longer term. This is important as sometimes the agent needs to make a bad decision in the short term, which maximizes the reward on the longer term. A simple reward function is not able to convey this information. The reward function is still required, as the value function depends on the reward function. The value function is defined as the expected return from the current state following a specific policy.

Policy

The word policy was briefly mentioned when defining the value function. The policy defines which action needs to be taken within a certain state to maximize the total reward. At first sight, one might think that the value function is defined for this. However, the value function is constructed based on the rewards received by trail and error. If the value function is not converged yet, this will lead to sub-optimal behavior. Should the agent purely follow the value function to receive the most reward, it would limit itself to perform actions which previously received high rewards, and would not discover potentially better actions. Several solutions are proposed for this such as ϵ-greedy and a stochastic policy. Both provide solutions by not always taking the current best action. This should improve exploration and improve the preciseness of the value function.

Exploration and Exploitation

Exploration and exploitation are one of the hardest parts of RL. As already explained, the policy should not only follow the value function as it might lead to a sub-optimal solution. The challenging aspect for an agent is to determine whether it should explore or exploit with its following action. Exploring results in a risk of obtaining less reward, but potentially discovering a higher reward. In contrast, if the agent acts greedily and exploits a certain action it already knows, the reward is certain. However, there is no possibility of obtaining a greater reward. The dilemma here is that neither exploration nor exploitation can be pursued exclusively without failing at the main task, which is achieving the highest reward. There are several options to handle this, several of them will be explained in the next sections.

2.1.2 Markov Decision Processes

A Markov Decision Process (MDP) provides a framework for decision-making. The framework of a MDP is in essence the same as provided in 2.1.1. In principle a RL problem can almost always be described as a MDP. The MDP is now explained. An important property on which the MDP is based, is the Markov Property shown in Eq. (2.1).

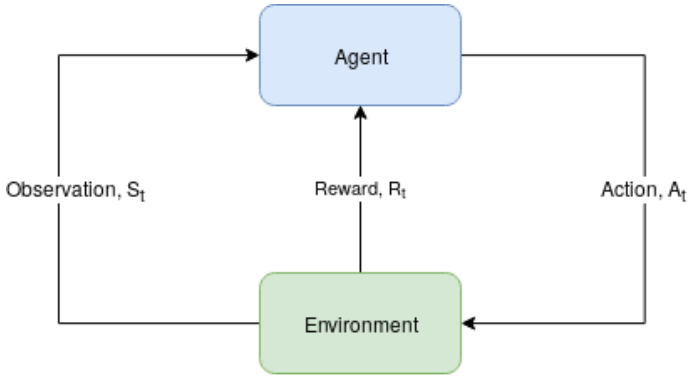


Figure 2.1: A schematic of a RL environment.

Definition 1: Markov Property

"The future is independent of the past given the present."

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ...S_t]. \tag{2.1}$$

A stochastic process has the Markov Property when the probability of ending up in a next state does not depend on history, but only on the current state. This may sound counter-intuitive as real-life decisions are often based on history. This can be resolved by containing all the important historical information within the current state. Defining processes this way simplifies describing decision models significantly, as all relevant information is provided in the current time step. The definition of the Markov property is given in Eq. (2.1). An example is provided to explain this concept properly. The environment is schematically represented in Fig. 2.2. In this environment, there is a start state, two termination states and multiple intermediate states, which together make the finite set of states S. In this example, the agent can opt for four different actions: Up, Down, Left or Right. Together this makes the finite set of A. The state transition probability matrix P, describes the chance the agent will end up in a certain state, given the current state and the action it takes. For this example, there is an 80% chance the action chosen, actually propagates the agent in this direction. A 10% chance the agent has a -90° deviation relative to the action taken, and 10% chance for a 90°deviation. The last important aspect is the reward function R, which is fairly simple. For each state, the reward will be 0, except for the two end states. The reward in the green end state is one, and the red end state minus one. The whole environment is now specified within a tuple, which defines the MDP. This tuple describes everything needed to make decisions in a certain defined environment. In Section 2.1.3 this MDP is used to explain dynamic programming.

Definition 2: Markov Decision Process

Markov Decision Process is a tuple, with $\langle S, A, P, R \rangle$

- S is a finite set of states
- A is a finite set of actions
- P is a state transition probability matrix
  $P^a_{ss'} = \mathbb{P}[S_{t+1} = s^{'}|S_t = s,, A_t = a] = T(s, a, s')$
- R is a reward function
  $R^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = R(s, a)$

2.1.3 Dynamic Programming

The MDP describes a framework for decision-making. However, the MDP does not define how to maximize the reward in the defined process. In the example, this would mean that the agent knows how it should move through the grid world to receive the most reward. When the agent arrives in one of the termination states the MDP ends. It then could start over to take another run. These sequences within an environment are called episodes. The next episode starts independent of the previous episode. Dynamic programming is a mathematical method to solve a MDP. Solving an MDP yields a complete description of which action to take depending on the current state to maximize the reward. Before dynamic programming is elaborated, the Bellman equation and the return are introduced.

Return

The goal of an agent is to maximize its return. The return G_τ is defined as the rewards obtained until the end of episode τ, starting from a specific state. This is stated in Eq. (2.2). Future rewards can be discounted by a $\gamma \in [0,1]$, which will be explained in the next section. The expected return following a policy π is the value function, which is stated in Eq. (2.3). The value function $V(s)$ defines how good it is to be in a certain state. This is useful because if the agent knows how good it is to be in a certain state, the agent could make decisions to propagate to a better state.

$$G_\tau = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T \tag{2.2}$$

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.3}$$

Bellman Equation

The Bellman equation describes the value function. The Bellman equation is stated in Eq. (2.4). The essence of this equation is that the value of the current state is described as the immediate reward, plus the expectation of all future rewards, given a certain action. This is the probability of all probable next states as the result of this action, times the corresponding value function of the next state. The γ discounts possible uncertain future rewards. As γ increases, the future rewards become increasingly important. The γ lies between $[0,1]$.

Definition 3: Bellman Equation

$$V(s) = \max_a (R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s')) \tag{2.4}$$

γ is a discount factor: $\gamma \in [0,1]$

For dynamic programming it is required that the whole MDP is known. Thus, the rewards and transition probabilities for each state must be known. In the example of Fig. 2.2, the reward at each state is zero, except for the end states. The trick for finding out the value function, is to use synchronous backups. Each state will be updated based on the value function of its successor state. As can be seen in the Fig. 2.2, all value functions from all the different states start at zero, except for the end states which have a non-zero immediate reward. In the next iteration, adjacent states have been updated based on the successor end states. In the next state, this propagates further. After enough iterations, the value functions will converge. When the value functions are converged, the agent can choose actions that will give the most reward. The actions the agent takes are determined by the policy. After each value function update, the policy is updated as well. Something to note is that if the value functions are not converged yet, the optimal policy could still be found. A problem with RL is that generally the MDP is not fully known, and thus dynamic programming does not provide a solution. In Section 2.1.4 it is explained how to deal with partially observable MDPs, or unknown MDPs.

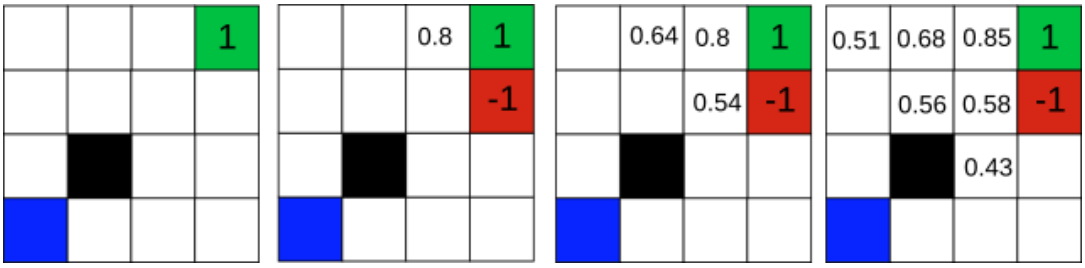


Figure 2.2: Simple example of value iteration.

2.1.4 Model Free Prediction

To specify a MDP, complete knowledge of the process is required. In real life problems this is often not the case. In a model-free environment, there is no knowledge about the states, state transition matrix or rewards. To solve these environments, dynamic programming can no longer be applied. RL can offer a solution. With RL algorithms, trial and error experience is backpropagated to the states. The best way to start to explain this is to start with the Monte-Carlo learning.

Monte-Carlo Learning

In Monte-Carlo learning, a state is evaluated by taking the mean of the return. This is an approximation to the value function, since the expectation of the return equals the value function. This is accomplished as follows: the agent visits random states and increases the counter of the corresponding state it visits. Each state thus has its own counter, which counts how many times the agents visited the state. These counters are not reset at the start of new episodes. If the episode ends, the return G_T for each state is calculated and added to the total return $S(s)$ which each state individually keeps track of. The value function is now calculated by dividing the total return of each state by the number of times the agent visited this state. With enough episodes, the value function will converge. This process is also described in Algorithm 1

Algorithm 1: Monte Carlo Learning

Result: V(s) = S(s)/N(s)
while *N(s) «* ∞ **do**
  for *each s in States* **do**
    if *agent visits state: s* **then**
      Increment Counter for N(s);
    end
    if *episode ends* **then**
      S(s) $\leftarrow$ S(s) + G_T;
      V(s) = S(s)/N(s);
    end
  end
end

The algorithm could be rewritten, such that the total return ($S(s)$ for each state is not required. Instead, the error between the current value function and the return is added, multiplied by one divided by the amount of visits the agent made. This is shown in Eq. (2.6). Eq. (2.5) still counts how many times the agent visited a certain stead. This procedure yields the same results as the method explained above. In Appendix A the derivation is written out.

$$N(s_t) = N(s_t) + 1 \tag{2.5}$$

$$V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}(G_T - V(s_t)) \tag{2.6}$$

The disadvantage of Monte Carlo learning is that only after an episode ends, the value functions of the states visited are updated. Thus, if for example there is no end state, Monte Carlo learning cannot be applied. This could be solved by Temporal-Difference (TD) learning. With TD learning, the states could be updated with incomplete episodes. Furthermore, TD learning generally learns faster, and reduces variance of the value function in comparison to Monte Carlo learning. The variance is reduced because TD learning updates the states more often. The next time step can directly benefit from this update, which also increases the learning rate. However, this is not always the case and highly depends on the applied environment. A disadvantage of TD learning is that a bias is introduced. This will be explained in the next section.

Temporal Difference Learning

Temporal-Difference (TD) learning uses a method called bootstrapping. Bootstrapping, is basically updating the value function with a guess. To explain this in more detail, first TD(0) is discussed. The zero in TD(0) will be explained later. To explain TD(0), first the Bellman equation is discussed. The Bellman equation states that the return of a certain state is equal to the immediate reward, plus the discounted value function evaluated at the next state. In TD this last term is substituted for the expected return compared to Monte Carlo learning, shown in red in Eq. (2.7). Since the value function might not be converged yet, this is a guess. Updating the value function by something that is still a guess might seem a weird approach. But the value function contains one value which is not a guess, which is the immediate reward received for the next state. After several updates this truth will converge the value function just as with Monte Carlo updates.

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.7}$$

TD learning can also guess multiple steps into the future. So for example 2 steps $R_{t+1} + \gamma V(S_{t+1}) + \gamma^2 V(S_{t+1})$. Or even n steps, where the remarkable thing is that, if $n = \infty$ this is the same as Monte Carlo updates.

To elaborate on this, there is an algorithm called TD(λ). TD(λ) combines all possible n steps, to one algorithm. To achieve this, all n are given a certain weight, where the total weight is equal to 1. The lambda defines how important each n is. If $\lambda = 0$, the algorithm is equal to TD(0).

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{n-1} \tag{2.8}$$

This combines the advantages of MC and TD learning, and surprisingly reduces the disadvantages of both algorithms. However, a problem which arises again is that the whole episode needs be completed before TD(λ) can be updated. Looking backward instead of forward can resolve this issue. This is done using the so called eligibility traces. In current algorithms TD(λ) is mostly replaced by an algorithm called Generalized Advantage Estimator (GAE) which gives similar results (Schulman et al., 2018).

2.1.5 Value Based

In this section it is explained how TD learning is used to do actual control on the agent based on the value function. Also, the Bellman equation will be rewritten to make it more useful in the context of RL. In value based control the policy takes the action that leads to more reward.

A problem that arises if the policy does that, is that the agent does not explore. The agent only follows what it "knows". To address this issue, the agent should also take random actions, not based on the value function. To implement this an algorithm called ϵ-greedy can be used. With ϵ-greedy there is $1-\epsilon$ chance the agent acts greedily with respect to the current value function and ϵ chance the agent acts randomly. If enough steps are taken the agent will figure out the optimal value function and should act only greedy from that point on. Some algorithms include a decaying ϵ such that this is accomplished.

The value function is rewritten to a Q function, which stands for the action-value function. In the Q function, actions are included in the function parameters. The current Bellman equation takes the maximum reward in which a certain action is involved. With RL we need to be able to take an action that is not always optimal to facilitate exploration. The Q function makes this possible. Therefore, the mathematical relation between the Q function and Value function is:

$$V(s) = \max_a(Q(s,a)) \tag{2.9}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \tag{2.10}$$

The Q function is thus the same as the Value function with the only difference that the action taken can now be chosen. The expected future reward is still the value function, as the maximum is taken. Q-learning and SARSA (Sutton and Barto, 2018) are both RL algorithms which use the action-value function. The Q-learning algorithm is shown in Algorithm 2.

Algorithm 2: Q-Learning

for *each episode* **do**
  Initialize S
  for *each episode step* **do**
    Choose A from S using policy derived from Q (e.g., $\epsilon - greedy$)
    Take Action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$
  end
  until S is terminal
end

2.1.6 Value Function approximation

With RL we want the ability to solve big problems. However, as an MDP becomes bigger, the set of states could quickly become too big to process. Furthermore, as the world itself is continuous, this essentially means an infinite number of states. Even if you could store all these states, learning with all these states will be slow. To resolve this, we need to look differently at the value function. The value function now represents the expected return for actual states. With the value function approximation, the state is a parameter of a function that approximates the value of the current state. A weight vector is added to the function such that it can approximately fit the true value function, which can be seen in Eq. (2.11). The state parameters are defined as observations of the current environment, sometimes they are also called features. Neural networks are popular functions which can be used to specify the value function approximation. With RL the weights are determined. The use of neural networks will be explained more elaborate in Section 2.2.

$$\hat{v}(s,w) \approx v_\pi(s) \tag{2.11}$$

Objective Function

To learn the right weights for the value function approximation $\hat{v}(s,w)$, an objective function is needed. The objective is to learn the true value function, this is specified in Eq. (2.12). For now it is assumed that we already know the true value function v_π. To learn $\hat{v}(s,w)$, J(w) should be minimized.

$$J(w) = \mathbb{E}_\pi[v_\pi(s) - \hat{v}(s,w)]^2 \tag{2.12}$$

$$\Delta w = \alpha(v_\pi(s) - \hat{v}(s,w)\nabla_w \hat{v}(s,w)) \tag{2.13}$$

If $J(w)$ is differentiable, gradient descent methods will help us go to this minima. The equation for this is shown in Eq. (2.13). Similar to TD learning, for learning the value function approximation $\hat{v}(s,w)$ the true value function v_π is replaced with the current approximation of the return G_t. This is shown in Eq. (2.14).

$$\Delta w = \alpha(G_t - \hat{v}(s,w)\nabla_w \hat{v}(s,w)) \tag{2.14}$$

2.1.7 Policy Based

A policy based on following the value function, such as a decaying ϵ-greedy algorithm, is not always optimal. Sometimes the value function approximation is not able to differentiate between different states because the provided observations to the value function are similar. So states could appear the same, but require a different action. Furthermore, sometimes the best policy is a random policy, e.g.: rock-paper-scissors. In these kind of problems the value function will not converge to a good solution. To resolve these issues, the policy could be random to a certain degree. This is a policy-based approach, which is using a stochastic policy, without a value function. A stochastic policy is able to define an action distribution for specific states. This means that a stochastic policy is able to learn in which states it is better to act random and when not. In order to define such a stochastic policy, the policy should output an action distribution. For this the policy will change to a function build up from certain parameters. These parameters are labeled as theta and allow to define the policy as:

$$\pi_\theta(a,s). \tag{2.15}$$

A policy-based approach is optimized gradually. This can be done with a policy gradient, which is explained below.

Policy Gradient

Just as with the value function optimization, there needs to be an objective to optimize the policy. In this case it is referred to the policy objective $J(\theta)$. There could be several policy objectives, but an example of a straightforward policy objective is to receive the maximum attainable reward per episode. Receiving the most reward per episode is equal to maximizing the value function following policy π, where the value function is defined as the expected return. This is shown in Eq. (2.16). The idea is that by sampling the environment, $\mathbb{E}_{\pi_\theta}[v] = \sum_x p(x)v(x)$ can be estimated.

$$\begin{gathered}\text{In episodic environments:} \\ J(\theta) = V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}[v] = \sum_x p(x)v(x)\end{gathered} \tag{2.16}$$

To maximize the reward per episode the policy objective should be maximized. This could be done by finding the θ which maximizes the policy objective function.

One could search in every direction of theta to determine whether or not the policy objective function increases. However, since a policy typically contains many parameters, searching in each direction is an exhausting approach.

Therefore, another approach is required, namely, Stochastic Gradient Ascent (SGA). For stochastic gradient ascent to work, the policy objective needs to be differentiable in θ, which can be seen in Eq. (2.17). This will give the direction $\Delta\theta$ the policy should shift to. Thus, the policy itself should be differentiable in θ. However, differentiating the policy means that the expected return cannot be obtained by the environment anymore. As this would result in Eq. (2.18). To be still able to take samples of the environment a clever trick as has to be applied (Meyer, 2016). This trick is shown in Eqs. (2.19) and (2.20) and is briefly explained below. More elaborate information can be found in (Meyer, 2016).

$$\Delta\theta = \nabla_\theta J(\theta) \tag{2.17}$$

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[v] \tag{2.18}$$

The gradient of the policy is the same as the policy times the gradient of the log of the policy, shown in Eq. (2.19). Due to this trick, Eq. (2.18) can be avoided. Instead the expectation can be taken again which can be seen in Eq. (2.21).

The gradient of the log of the policy is called the score function, this is shown in the right side in Eq. (2.20). The score function determines how to increase the frequency of a particular actions. By multiplying this by the return of the particular action, one finds the direction in which θ should be changed in order to increase or decrease the frequency of the particular action. The satisfying thing about this trick, is that the expectation can still be taken. As the multiplication of the policy can be substituted by taken the expectation. As already mentioned, this is needed because now samples can once again be taken from the environment.

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \tag{2.19}$$

$$= \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a) \tag{2.20}$$

Definition 4: Policy Gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) G_T] \tag{2.21}$$

An example of an algorithm that uses policy gradients is the REINFORCE algorithm, which is shown in Algorithm 3. The algorithm includes an alpha. The learning rate is determined by α, thus α determines how much θ is changed.

Algorithm 3: REINFORCE

for *each episode* **do**
  for *t=1 to T* **do**
    $\theta \leftarrow \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) v_t$
  end
end

2.1.8 Actor Critic Based

The actor critic based solution combines the solutions of the value based and policy based approach. As the algorithms name suggests, in this approach there is an actor and a critic. The actor is the part that uses the policy gradient algorithm and the critic the value based algorithm. The critic supervises the actor and communicates whether an action is beneficial or not. The critic calculated a Q value which is passed to actor to update the policy. This can be

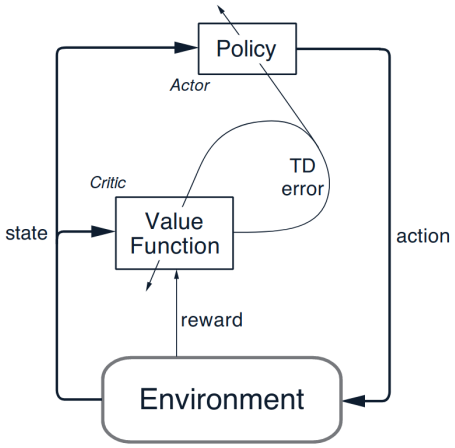


Figure 2.3: Actor Critic (Sutton and Barto, 2018).

seen in Eq. (2.22). This essentially means the TD error is propagated to both the policy and value function approximators. This is shown schematically in Fig. 2.3.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_T] s \tag{2.22}$$

- Critic updates w by value TD methods
- Actor updates θ by policy gradient

An improvement for updating the stochastic policy could be made by using the advantage instead of the Q value. The advantage is the difference between the state-action value function and the value function, which is stated in Eq. (2.23). The advantage defines how much better or worse an action is compared to the value of the current state. The convenience of using the advantage over using the Q value, is that it lowers the variance of policy updates.

For example, suppose the reward of a specific action within a specific state is 1001, and from another action it is 999. Both rewards are positive rewards, but the relative difference is important here. If the current value function states that a reward of 1000 is expected. The first action exceeds the value function with 1. The other action under performs according to the current value function. This actually means that the second action should be avoided, and the weights should be decreased. However, if a Q-value of 999 is supplied in Eq. (2.21), the weights would be increased. Changing the Q-value to the advantage will actually decrease the weights. Furthermore, as the only the relative difference between the Q-value and the current value is taken. The updates will be relative to the current value function. The same weight updates will be applied for rewards received such as 501, with a current value function of 500. This means that using the advantage lowers the variance.

$$A(s,a) = Q(s,a) - V(s,a) \tag{2.23}$$

2.1.9 Policy Optimization

The learning rate is an obstacle for policy gradient algorithms. As previously mentioned, in Section 2.1.7, the learning rate in Algorithm 3 determines how much the policy parameters θ change. If the learning rate is huge, learning can become unstable. But if the learning rate is too low optimizing could take a long time. There are various policy gradient algorithms to improve

the algorithm such as: Sample Efficient Actor-Critic with Experience Replay (ACER), Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO). PPO will be treated here since PPO (Schulman et al., 2017b) it is the algorithm used later on in the experiments. Moreover, as PPO is based on TRPO, TRPO is briefly introduced as well (Schulman et al., 2017a; Hui, 2018).

Both algorithms rely on keeping the updated policy within a certain boundary. Thus, a new policy does not diverge excessively from the old policy (Schulman et al., 2017b). In contrast to a normal policy gradient algorithm there is a boundary on the policy distribution and not on the parameter space. A small change in policy parameters θ, could lead to drastic change in policy distribution, which might lead to a drastic drop in performance. Both TRPO and PPO try to avoid this. Both algorithms try to maximize Eq. (2.24). Eq. (2.24) describes the performance relative to the old policy. The action is more probable in the current policy compared to the old policy, if $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ is greater then one. If it is less then one, the action was more probable in the old policy θ_k.

$$J(s,a,\theta_k,\theta) = \mathop{\mathbb{E}}_{s,a\,\pi_{\theta_k}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(a,s)\right] \tag{2.24}$$

TRPO bounds the new policy in terms of KL-Divergence. This measures a "kind" of distance between two policies.. TRPO maximizes its objective function the following way, where D_{KL} is the KL divergence. The objective given function is bounded by δ.

$$\begin{aligned} \theta_{k+1} &= \arg\max_\theta J(\theta,\theta_k) \\ s.t. D_{KL}(\theta||\theta_k) &\leqslant \delta \end{aligned} \tag{2.25}$$

PPO is based on the same principle of TRPO, but has less complex mathematics which make the algorithm faster and simpler to implement. PPO has 2 versions, PPO-Clip and PPO-Penalty. Here PPO-Clip is treated as this is the one used in the project. The advantage of PPO-Clip is that is doesn't use a form of KL-Divergence but instead only relies on clipping the policy itself. Eq. (2.26) shows how this is done, where ϵ is a parameter which defines by how much the policy is clipped.

$$\begin{aligned} &\text{If advantage is positive:} \\ &J(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a,s)}{\pi_{\theta_k}(a,s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(a,s) \\ &\text{If advantage is negative:} \\ &J(s,a,\theta_k,\theta) = \max\left(\frac{\pi_\theta(a,s)}{\pi_{\theta_k}(a,s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(a,s) \end{aligned} \tag{2.26}$$

When the advantage is positive the objective function will increase if the $\pi_\theta(a,s)$ becomes more likely. However, the min operation will prevent that the objective function can increase too much. This way a policy which diverges a lot from the old policy is prevented. It can only increase by: $\pi_{\theta_k}(a,s)(1+\epsilon)$ When the advantage is negative the objective function will decrease if the action becomes less likely. This will thus happen when $\pi_\theta(a,s)$ decreases. The max ensures that it can only decrease by $\pi_{\theta_k}(a,s)(1-\epsilon)$.

2.1.10 Summary

In this section a small summary is given to explain the gradual build up of the Proximal Policy Optimization (PPO) algorithm of this chapter. In Fig. 2.4 there is an illustration of the Value-Based, Policy-Based and Actor Critic RL architectures. The basis of these architectures is still

the basic RL schematic in Fig. 2.1. As explained in previous sections, the actor critic approach is a combination of a policy-based and value-based approach, this is also illustrated in Fig. 2.4 (4). In the figure the Value-Based approach is illustrated in (3). The agent consist of the value function and the epsilon-greedy algorithm. By using the epsilon-greedy algorithm, the agent takes the action with the maximum reward or it takes a random action. The value function is updated by the temporal difference error. The exact update is shown in the equation of (3). In (1) the policy-based algorithm is shown. Here the actions are taken based on a stochastic policy. The stochastic policy is created by combining a neural network with an action distribution. The neural network is updated by the received return. This is also shown in the accompanying equation. At last, in (2) the architecture of the actor-critic is illustrated. Here there is stochastic policy that uses the value function to update the stochastic policy. This can be seen in the accompanying equations. Using the advantage instead of the Q value is an extension of the algorithm.

2.2 Deep Reinforcement Learning

The policy π_θ and the value function V_ϕ are introduced in the previous algorithms. As already mentioned, both consist of parameters θ and ϕ which need to be modeled. Although this can be done with simple linear functions, most policies and value functions need complex representations. Artificial Neural Network (ANN)s are powerful function approximators which haven been successfully used in several areas such as image recognition, self driving cars and robots. The combination of ANNs and RL is called Deep Reinforcement Learning (DRL).

An ANN is inspired by biological neural networks. This framework is used in many machine learning algorithms to design complex functions which are able to process complex data. An ANN consists of multiple neurons which are also referred to as units. Each unit can send a signal to other units. In a unit, all inputs are multiplied by the associated weights and summed up, afterwards a bias is added. After the summation, an activation function is applied to the value found. This is mathematically represented in Eq. (2.27).

$$Y = \mathrm{a}(\sum(\text{weight} * \text{input}) + \text{bias}) \tag{2.27}$$

2.2.1 Activation Functions

An activation function kind of determines how important a value from a certain unit is. It does this by determining in what way the value should be passed through or not. The simplest activation function for this would be the step function. The step function outputs a one if its input is greater than zero, or a zero if the input is smaller than zero. This could lead to drastic changes in the output by minor changes in the input. Also, the step function has a derivative of zero everywhere except at 0 where it is infinite. This makes changing the weights by back propagating difficult, as the weight should be gradually changed. This is further explained in Section 2.2.3. The sigmoid function looks like a gradual step function, where its derivative is more gradually. Using the sigmoid, or any other nonlinear function, allows the neural network to capture nonlinear relations. A linear activation makes multiple layers in the ANN useless, as the composition of several linear functions is still a linear function. Also, several machine learning tasks cannot be solved if the output of the ANN is linear to the input. There are several activation functions, but the most popular functions are the sigmoid Eq. (2.28) and tanh Eq. (2.29) and relu Eq. (2.30) functions.

$$\text{sigmoid:}\quad a(x) = \frac{1}{1+e^{-x}} \tag{2.28}$$

$$\tanh:\quad a(x) = \frac{2}{1+e^{-x}} - 1 \tag{2.29}$$

$$\text{relu:}\quad a(x) = \max(0, x) \tag{2.30}$$

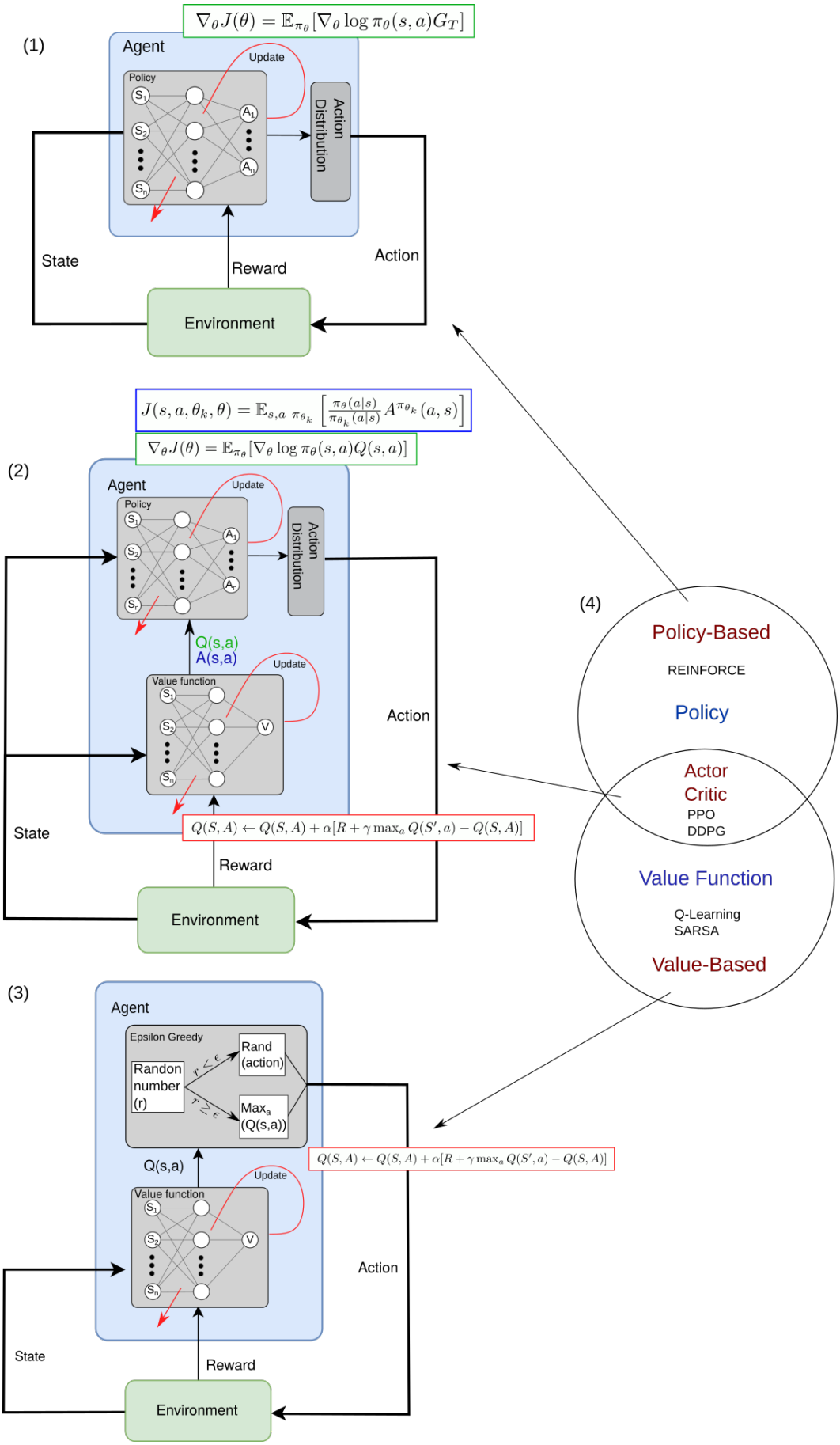


Figure 2.4: Summary illustration of the background. Containing the architectures for the Value-Based (1), Policy-Based (3) and Actor Critic (2).

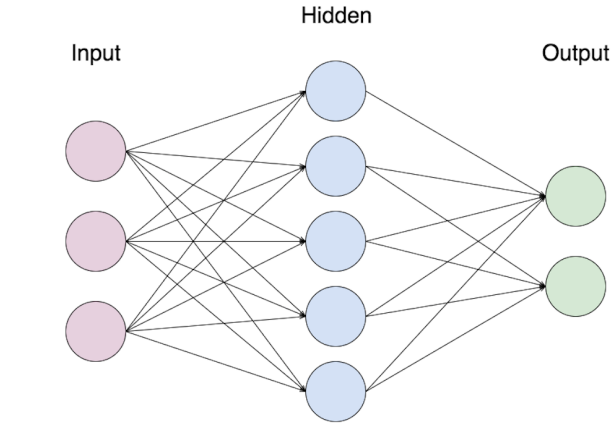


Figure 2.5: Simple Neural Network.

2.2.2 Layer

An ANN consists of multiple units arranged in layers. The simplest configuration is an input layer, a hidden layer and an output layer. However, multiple hidden, input and output layers are possible as well. Furthermore, different types of layers are available. Fully connected layers are most commonly used. As the name suggests, a unit in a fully connected layer is connected to every unit in the previous layer. An example of ANN is given in Fig. 2.5. Some other types of neural network architectures are the Convolutional neural network (CNN) and Long Short-Term Memory (LSTM). The Convolutional neural network (CNN) is primarily used in image recognition. The Long Short-Term Memory (LSTM) could be used for sequence observations. These architectures can be combined in several ways.

2.2.3 Backpropagation

In Section 2.1.6 the objective function is introduced, which defines a way to update the weights of the value function. The objective function is sometimes also called the loss or cost function. A way of representing the policy and value functions is by using a neural network. Something that was not explained is how to differentiate a neural network such that the objective function can be optimized. To update each node in the correct correction direction to minimize or maximize the objective function, the effect of each node on the output has to be determined. This is done by an algorithm called backpropagation (Fei-Fei et al., 2017). With backpropagation the gradients relative to the output of each node in the network can be determined. This is done by determining the local gradients of each node first. With the local gradients, the real gradients of the nodes can be calculated by the chain rule. A more elaborate explanation can be found here (Fei-Fei et al., 2017).

2.3 Hierarchical Reinforcement Learning

As already mentioned, RL is based on how humans learn. However, humans are able to use previous skills to solve new problems. Hierarchical Reinforcement Learning (HRL) tries to mimic this. Almost every high level task can be divided into several sub-tasks. Previously mentioned RL models, are only capable of learning one specific thing. This makes learning complex tasks much more difficult, as the rewards for this problem will probably be sparse. HRL is an approach to improve and attend several issues which RL suffers from, by implementing a hierarchical approach. Some HRL frameworks are treated now.

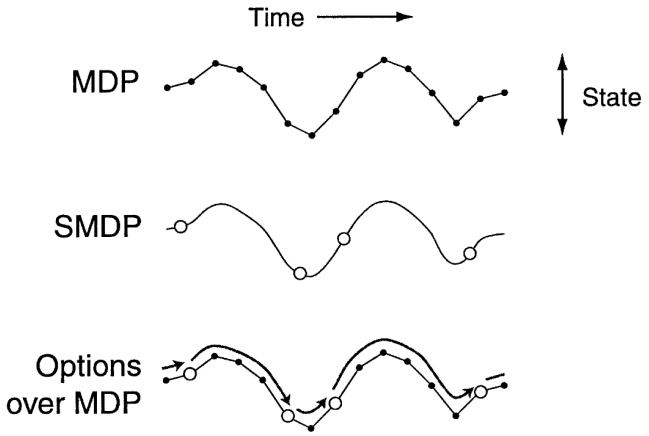


Figure 2.6: Semi Markov Decision Process (SMDP), Sutton et al. (1999).

2.3.1 Semi Markov Decision Process

In RL a MDP is solved. To extend RL to HRL, the MDP needs to be extended to the Semi Markov Decision Process (SMDP) (Sutton et al., 1999). In a SMDP the time length for each action taken could differ, where the MDP has a fixed time length. This can be seen in Fig. 2.6. Although the time length of each action to state can now differ, the theory used above is still applicable.

2.3.2 Option Framework

Due to the SMDP, macro-actions also referred as options can be defined. Options exist out of several actions in a sequence. This sequence is determined by its own policy, thus the actions do not have a fixed sequence. In the original option's framework each option also has a termination condition and an initiation set (Sutton et al., 1999). Options are defined as a tuple containing: $\langle I, \pi, \beta \rangle$. Where, I is the initiation set, π the corresponding policy and β the terminating condition. The initiation set determines whether or not an option is available in the current state. The terminating condition determines when the option should end. Both the terminating and invitation set restrict the utilization in a potentially proper way. As can be imagined, not all options should be used under all circumstances as they are specialized in achieving a certain inner reward and goal. So they should end when a certain goal is reached. Which option is used at which time is determined by a master policy. Changing the action-value Q function, to an option-value function is straightforward, as the action could just be replaced by an option, shown in Eq. (2.31). The top policy is not essentially restricted by only choosing options. It could also easily choose actions, as for the top policy actions and options essentially do not differ. There are several variations of this framework possible. In the framework of Frans et al. (2017) they reset the master framework when a new task is completed. The sub-policies are not reset, and should generalize over time as different tasks are completed. More elaborate information is in the paper.

$$Q(s,o) = R(s,o) + \gamma \sum_{s'} T(s,o,s') \max_{o'} Q(s',o') \tag{2.31}$$

2.3.3 Feudal Network

FeUdal Networks (FuNs) are another framework to incorporate HRL (Dayan and E. Hinton, 1993). In feudal learning there is a hierarchy in the form of super-managers, managers and sub-managers. The managers work for the super-managers where the sub-managers work for

the managers. The difference with the option framework here, is that managers could give sub-managers subgoals. The manager gives a reward if this sub-task is achieved, even if the accomplished sub-goal did not aid in the managers own goal. This hierarchy drives that the managers higher up the pyramid, automatically define higher level tasks. Sub-managers will figure out how to accomplish lower level tasks. Managers have two choices, namely, which specific goal a sub-managers should fulfill, and which sub-manager it should fulfill.

2.4 PIRATE

The PIRATE is a Pipe Inspection Robot for AuTonomous Exploration which is under development at the research group Robotics and Mechatronics (RaM) of the University of Twente. The current design of the PIRATE exists out of seven sections as can be seen in Fig. 1.2. As the pirate is symmetric it can be separated into a front and back section. The front and back consist of three sections which are connected with revolute joints. As can be seen in Fig. 1.2 one of the maneuver's the PIRATE could make is clamping the back and front. A wheel is located between each of the parts, which all have a DC motor. In the middle there is a rotatable section such that the PIRATE could rotate its back or front section, depending on which part is clamped.

2.5 V-REP and PyRep

V-REP[1], is a robot simulator. In V-REP it is possible to distinguish between visual and dynamic shapes. In V-REP each object/model can be controlled by a script. There are several ways to implement this, which makes V-REP versatile. Examples of possible implementations are: embedded scripts, and internal scripts based on Lua, plugins which can be written in C/C++. There is a remote API client, which is available in several languages. Furthermore, ROS and Bluezero are supported as well. Each implementation has its advantages and disadvantages. The interesting part for this thesis is that V-REP supports object and model control by Python via the remote API. As most of the RL development is built in Python, support for Python seemed a necessity for this research. The disadvantage here is that the Remote API is really slow. The solution is PyRep, PyRep is a plugin on the open-source project V-REP which removes several latencies (James et al., 2019).

2.6 Ray

Ray is an open source framework which consist out of three libraries: Tune, RLlib, RaySGD. Two of these libraries, Tune and RLlib, are treated here as they are used in the thesis. RLlib is a open-source python library focused on providing a scalable reinforcement learning framework. Tune is also an open-source python library which focuses on scalable hyperparameter tuning. RLlib is highly customizable. Furthermore, it provides high level components such that a reinforcement learning project could be easily started. The main structure of RLlib is specified as in Fig. 2.7. Most of these components are easily user customizable. But as the library is open source even these can be customized.

The most important components of RLlib are the trainer, policy, model, action distribution and environment. These will be shortly explained, starting with the environment. Several environments are supported by RLlib[2]. Among these, the well known OpenAi Gym environment is supported. Additionally, multi agent and external agent environments are supported. The policy defines how the agent acts in the environment. In the policy component, the loss, action distribution and model are defined. The model defines the policy π and value function V_ϕ. Implementation support for TensorFlow or PyTorch is integrated. The action distributions component defines how the action distribution is defined based on the model output.

[1] During the writing of this thesis a new version came out called CoppeliaSim (Rohmer et al., 2013). However in this thesis V-REP is used.

[2] https://docs.ray.io/en/latest/rllib-env.html

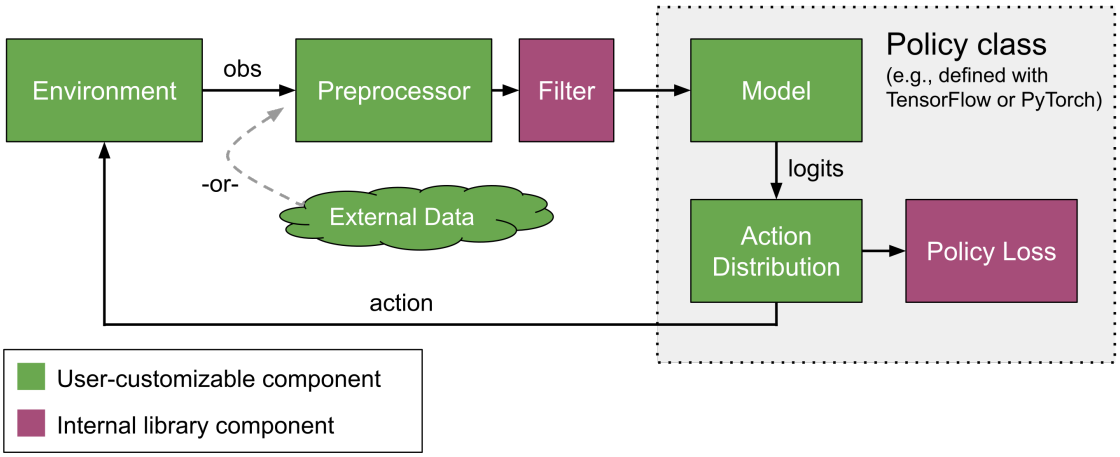


Figure 2.7: RLlib implementation schematic (Anyscale, 2020).

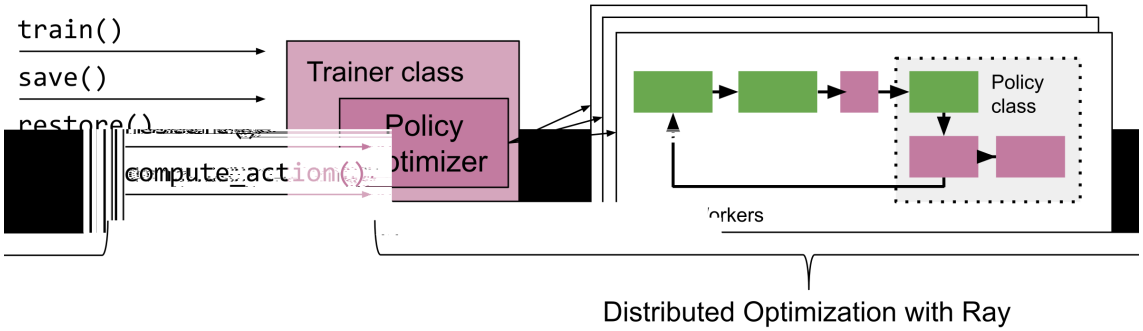


Figure 2.8: RLlib main components wrapped in rollout workers controlled by a trainer class (Anyscale, 2020).

The RLlib components of Fig. 2.7 are wrapped within certain amount of so-called rollout workers. This is shown in Fig. 2.8. The rollout workers then collect batches of observations and rewards.

In RLlib several RL algorithms [3] are available. These algorithms are available through trainer classes. These trainer classes setup all the other components necessary to train an agent. Such as the algorithm configuration and setup of the workers. They also control the rollout workers. This is shown in Fig. 2.8.

Tune is a library which can be used on top of RLlib (Liaw et al., 2018). This library is used to tune hyperparameters. Tune integrates with several optimization libraries such as HyperOpt, Bayesian Optimization and Facebook Ax. On top of this, several algorithms to scale the search are implemented such as Population Bases Training (PBT).

[3] https://docs.ray.io/en/latest/rllib-algorithms.html

3 Analysis and Methodology

In this chapter the analysis and methodology of this research are explained. In Section 3.1 the problem is analyzed and the simplifications made in this research are explained. Section 3.2 outlines the basic principles of the RL model and the final choice of the RL algorithm of this research. Moreover, a subsection is devoted to how one should choose appropriate hyperparameters. The hyperparameters of the RL algorithm used highly affect the outcome of the training. Section 3.3 deals with Hierarchical Reinforcement Learning (HRL). Because this section is based on results and methods of the previous sections, it contains both a separate analysis and methodology. This section is based on the results of the experiments performed from the previous sections in the Chapter 3.

3.1 Problem Analysis

As previously mentioned in the Chapter 1, the goal is to let the PIRATE autonomously navigate in a realistic 3D simulated environment by RL. The simulation environment used is V-REP, which is discussed in Chapter 2. The PIRATE will be spawned at the beginning of a pipe system in the simulated environment. In this section the problem is analyzed. To properly analyze the problem, some simplifications are first explained.

3.1.1 Simplifications

There is a direct relation with the problem difficulty and the design of the pipe system. The more complex the pipe system, the more difficult the problem will become. The pipe system designed for this thesis is slightly simplified, but still interesting to perform experiments on. First of all, the pipe segments always have the same diameter. Furthermore, all bends are 90° and after each bend there will be a straight pipe section. In a real pipe system, more complicated configurations might be present. For example, a real pipe system might contain T-sections, several different varieties of bends, pipe sections with different diameters and combinations of these elements. Additionally, as there are no T-sections present in the simulated environment, the PIRATE does not have to choose which direction to take.

Furthermore, simplifications are also made on the constructed model of the PIRATE. The physical PIRATE has springs between the bending modules. The advantage of this is that it is able to lock its joint position and still apply torque by springs when it is clamped within a pipe section. Otherwise the DC motors have to constantly apply torque which is not energy efficient. The constructed PIRATE model does not contain springs, as the torque can be applied by the DC motors in the simulation. Applying this to the model would significantly increase the complexity of the model. Another thing that is simplified, is that the rear module is removed. The rear module is still shown in Fig. 1.2. This is removed since it has no direct function and is not actuated. In the physical PIRATE the rear section serves to store several electronic parts and the connection cable. This cable powers the PIRATE such that batteries are not constantly required for testing.

Additionally, the exact friction, mass and torque of the robot are not taken into account. Instead default or carefully selected values are chosen, which are based on achieving realistic and stable behavior of the PIRATE in the simulation. Using the corresponding physical model properties of the PIRATE is out of the scope of this thesis.

3.1.2 Analysis

All simplifications are made such that the current problem becomes more manageable. Although this makes the problem less complex, it is still challenging. The PIRATE needs to solve

several challenging problems, such as entering and leaving a turn, climbing in a pipe and performing combinations of certain tasks. Furthermore, it has to do this in an environment which has only a few features. This is because from within the pipe system, it looks approximately the same from every point.

Suppose that the reward structure is solely based on the PIRATE driving itself forward. A requirement for driving itself forward, is that the PIRATE is properly clamped. With this specific reward structure the PIRATE does not receive a reward if it just clamps its joints. However, clamping is essential in order to drive forward. This makes driving forward based on this reward function a difficult task, since only the combination of clamping and driving the wheels in the correct orientation is rewarded.

By using a reward which is only based on driving itself forward, the PIRATE receives sparse rewards and this makes learning difficult. This could be solved by adding rewards for certain sub-goals. For example, by giving a sub-reward if the PIRATE is clamping. However, adding these sub-rewards for certain sub-goals might be harmful, as the PIRATE is then incentivized to perform the actions linked to the sub-rewards, which restricts its freedom in figuring out a solution for the main goal. However, by omitting these sub-rewards the PIRATE has to discover that it has to perform a combination of various actions at the same time solely based on a reward for driving forward. This becomes even more challenging as the world is continuous. In the simulation multiple actions are carried out every second. If the PIRATE wants to change its joint position several actions should be executed in a correct sequence. Clarifying this, if joint actions are changed too often the joints will go back and forth and never reach new end positions. As these actions are random in the beginning, it takes some time to discover how it should clamp.

As there are not a lot of features available in the pipe, it is important to supply proper observations, so the PIRATE can distinguish between several situations. For example, joint-positions and orientations could help in determining if its in a corner, and in which direction it is going. Probably the hardest thing for the PIRATE to accomplish is to learn how to enter a turn, and especially how to do the preparation for making a turn. What is meant by this, is that the front part of the PIRATE should already be aligned with the direction of the turn. If it is wrongly oriented it will be exceptionally difficult to still take the turn. Camera images could indicate an upcoming corner, such that the PIRATE knows how it should orientate itself. This situation leads to similar problems as encountered when the PIRATE learns to clamp. Preparing for a turn does not give immediate reward, as only actually going through a corner will give reward. The reason the PIRATE has to prepare for a corner is because when the camera is really close to the turn, obvious features of the turn may disappear simply because the camera is too close. Because of this, the PIRATE could get stuck because only current observations are used to determine the next action. The value function should be able to incorporate possible feature rewards, and based on this figure out, that it has to prepare. Relying purely on the value function could be insufficient, introducing memory could help in navigating through corners. Previous camera images then could be used to determine the current action. Also it remembers certain performed actions increased its current reward. Memory can be applied by serving previous observations to the neural network. Another way would be by using Recurrent Neural Network (RNN) architectures, in specific LSTM layers. LSTM layers are neural network layers which to some extent store previous observations within the network itself.

The current RL algorithm has to solve the whole problem. With Hierarchical Reinforcement Learning (HRL) sub-policies could be used to solve smaller sub-problems. Entering a turn could then be handled by a specific sub-policy, which receives rewards specifically for this task. Then the master policy only has to figure out which sub-policies it should select. These sub-policies then behave as macro-actions. Only choosing from sub-policies might restrict the PIRATEs freedom in figuring out a solution, as the sub-policies are optimized for a specific

task. This would create the same effect as rewarding for sub-tasks. The difference here is that the master policy is able to switch between these sub-policies, where rewarding for sub-tasks would be simultaneously and constantly. Furthermore, the master policy is not restricted to choose merely from macro-actions. The master policy could be designed in such a way that macro-actions are implemented alongside regular actions. This approach creates structure, while still allowing for the maximum freedom in finding the solution.

3.1.3 Reward Structure

The goal in RL is in direct relation with its reward structure. The goal for the PIRATE is to drive through a pipe system. Many RL problems deal with finding and reaching a target (Franceschetti et al., 2018). In this case, the end of the pipe could serve as the target to reach. When a target is used to construct a reward function, the current distance between the target and the agent is commonly used as the reward. Generally, the Euclidean distance is used for this. In the environment used in this thesis this could form problems. As the PIRATE is confined to the pipe system, the Euclidean distance between the PIRATE and the target might first have to increase, before the PIRATE reaches the target. The reward structure then would punish the PIRATE for actually making progress to the target. Thus, the Euclidean distance is not a feasible solution for more complex pipe systems. This could be solved by defining the distance between the PIRATE and the target, as the distance through the pipe system. To accomplish this, information of the pipe system is required. In real life, for some older pipe systems this may not be available. Also, if information of the complete pipe system is available, this could be supplied as an observation which makes the RL problem easier. For these reasons is assumed that information of the pipe is not available. Eventually, figuring out the layout of the pipe can be done by incorporating Simultaneous Localization and Mapping (SLAM) algorithms. Some work on this has already been performed (Kumar, 2019). However, this is out of scope of this thesis. Another solution would be to set targets which are directly visible for the PIRATE, either by lidar, depth or visual inputs. A similar approach is demonstrated in (Zeng, 2019), which was mentioned in Chapter 1. When the PIRATE reached the target, the next target could be placed. However, one suspected problem that might occur is that the agent could get stuck if a target is not conveniently placed. Furthermore, the reward structure is less straightforward as future targets should be taken into account, such that the received rewards do not suddenly decrease. This last approach might work, however using a different approach might simplify the reward structure. The reward functions discussed currently are based on reaching a target. In the current environment the need for reaching a specific target is actually not there. The goal is just driving forward through the pipe system. The reward function could thus also be specified for driving forward. This can be realized by checking the previous position and comparing this to the current position. As the as the PIRATE only moves forward, this is a simple and probably effective solution. This is clarified in Section 3.2.4.

3.1.4 Observation Space

Defining a good observation space for an agent is important for figuring out a good RL solution. The observations are the inputs of the agent which define the actions and the value function. The PIRATE has several properties which can provide observations. These will now be discussed. The PIRATE consists out of multiple sections, joints and wheels. Each of these objects have several properties which could be beneficial observations. First, observations which seem required are considered. Afterwards, observations which are less critical, but possibly interesting are discussed.

One of the first observations one might think of are the rotational joint-positions. The rotational joint-positions are the direct feedback for actions which actuate the joints. The rotational joint-positions determine to some extend the posture of the PIRATE. The PIRATE being

aware of the current posture is important for discovering effective actions. However, rotational joint-positions are not enough to determine the posture in absolute space. The joint-positions in absolute space contribute as well. The joint-positions in absolute space give feedback about how joint rotations affect the other connected joints. Together they are able to determine the posture of the PIRATE. Furthermore, the joint-positions in absolute space give a reference for the position the PIRATE is located. One last important observation is the orientation of the PIRATE in absolute space. The orientation is important for recognizing that the rotatable section of the PIRATE is able to change the orientation between front and back. Also, it helps in figuring out how the PIRATE is orientated within the pipe. This is important for being able to drive upside down. The wheel velocities for example, need this observation to determine which wheels are attached to the ceiling of PIRATE. The currently mentioned observations, in combination with the camera observations, seem crucial. The PIRATE should be able to distinguish several situations with these observations.

There are however more observations, which could improve the performance of the PIRATE driving through the pipe system. These observations are the wheel torque, wheel velocity, joint torque, joint velocities, absolute joint velocities and observations of previous time steps containing states, actions and rewards. The directions of the wheel velocities determine in which direction the PIRATE is moving. Using the wheel velocity as an observation could be helpful since determining the correct wheel velocity is not so straightforward. For example, a wheel which makes contact to the ceiling needs a reversed velocity to drive forward. The optimal wheel velocity depends on multiple observations. This is because the wheel velocity to drive forward is dependent on the current posture of the PIRATE. So although the PIRATE should be able to figure out the correct actions based on the current posture of the PIRATE, direct feedback of these wheel velocities could be helpful. Observing the amount of torque required could give some extra insights in the performance of the PIRATE, such as getting stuck. This could help as the DC motors start to stall when joints are getting stuck. Knowing the current action and position is not enough information to predict the next joint position. Among other things, the velocity is also required to predict the next position. For this reason joint velocities could provide information such that the next joint-position could be better predicted. The same could be stated for absolute joint velocities, which provide information about the movement of the PIRATE in absolute space. As the wheels also move the PIRATE, the velocity of a joint cannot be purely be determined by joint-velocities. Furthermore, observations of previous time steps could help in predicting future states, this is especially helpful as the actions are continuous. Architectures such as the RNN could take care of this. Several experiments are done to check the effectiveness of certain observations.

In the simulation environment it will be easy to obtain absolute positions. Acquiring positional data of the PIRATE in the real world is quite difficult as, for example, using wheel encoders will probably not work. Some wheels will slip or not even touch the pipe, for these reasons it will be difficult to obtain positional data. The problem could maybe be solved by using combinations of LiDAR, visual inputs and wheel encoders. Furthermore, solutions based on advanced state estimation using SLAM algorithms could work (Rufus et al., 2020). In this thesis is assumed that these positions can be acquired.

3.1.5 Camera Analysis

As mentioned, the PIRATE should be able to use visual input to detect upcoming corners. There are basically two options for this visual input. Depth information or visual information. Based on the pipe like features, visual information will probably be less informative then depth information. The features of the pipe system will be clearer with depth information as visual images will look even more similar. The most important feature the camera has to detect is the direction of a corner. Depth information incorporates this information as a large gradient in the

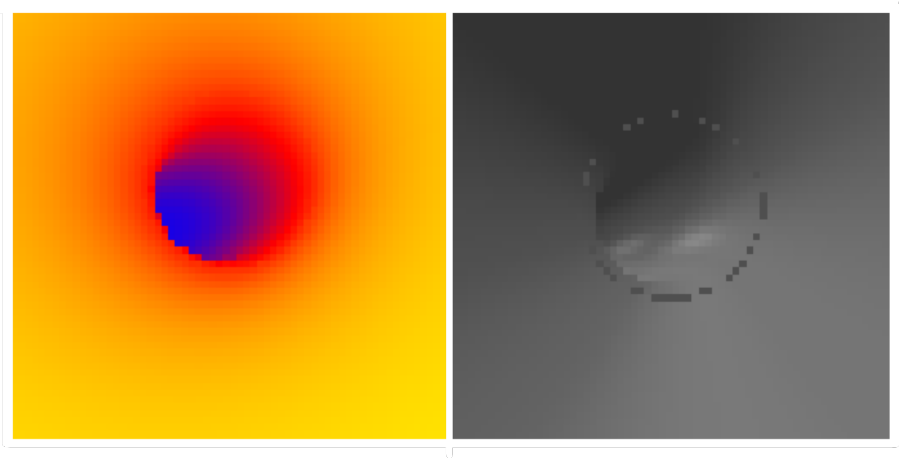


Figure 3.1: V-REP visual sensors. The difference between the depth and RGB (right) camera (left) is shown. The image resolution is 64x64x3.

direction of the corner. This gradient is also present in the visual image, however this might be harder to detect. This is shown in Fig. 3.1. Furthermore, a visual image depends on light in the pipe system, which is not present as the pipes are not transparent. A flashlight can be attached to the PIRATE such that light is present, however this could effect the quality of the visual image. A depth camera does not depend on this requirement.

The direction the camera will detect, is relative to its own frame. In principle this should be enough as the absolute orientation in the world frame should not matter. Several orientations will be provided to the observation space such that this connection can still be made. This can be helpful to deal with the effect gravity has. The PIRATE should be able to orientate itself properly for the turn, due to the effect of gravity different action responses are required.

As the images can contain a lot of data points depending on the resolution, the data should be pre-processed. Image processing such as convolution in combination with pooling and stride can help in discovering features, while also reducing the size of the data. Another approach of reducing the size of the data is by using a lower resolution image. A low resolution image might contain enough information.

3.2 Methodology

In this section possible RL algorithm are discussed as well as the corresponding hyperparameters.

3.2.1 Proximal Policy Optimization

The authors of Islam et al. (2017) compare two state-of-the-art RL algorithms for continuous control, namely Trust Region Policy Optimization (TRPO) and Deep Deterministic Policy Gradient (DDPG). They state that due to the amount of hyperparameters that need to be tuned it is difficult to compare results. They both suffer from performance variety if hyperparameters are changed. Also, it matters which environments are used and which performance metrics are looked at. However, in the compared environments TRPO seems to outperform DDPG.

Furthermore, Henderson et al. (2019) compare several algorithms, including DDPG, PPO and TRPO. It seems that trust policy algorithms perform more stable in less stable environments. Of course, the environment in this thesis is not directly comparable to the environments used in that research, however driving vertically up the pipe could give some unstable situations. They also note that hyperparameters and tuning highly influences the results.

In this research the RL algorithm used is Proximal Policy Optimization (PPO), which is based on TRPO. PPO is the algorithm considered in this thesis as it is easier to implement and tune (Schulman et al., 2017b). Although PPO is chosen, it does not mean that the performance of the PIRATE could not be better by using DDPG or TRPO. Evaluating this could be something for future work.

3.2.2 Actions

As already mentioned, the PIRATE has six wheels and six revolute joints. Together this gives the PIRATE twelve possible control actions. V-Rep has two options, velocity control or position control. Position control is done by modulating the velocity of the joint, using a PID controller. Maximum torque and velocity could be set in the configuration of the joint. Either way, the PIRATE is essentially controlled by velocity.

Controlling the wheels is obviously done by velocity control, as it is not logical to do this by position control. Especially because the velocity of the wheels can be set by a discrete action, as there should not be an advantage of using a continuous velocity for the wheels. This simplifies the output space, which has the advantage that the PIRATE should be learning faster. The possible actions for one wheel are $(4, 0, -4)$, in rad/s.

All the joints of the PIRATE are controlled by continuous velocity control, except for the rotatable and camera section. The reason velocity control is used, is because this makes figuring out how to clamp easier. The actions space is defined between $[3.14, -3.14]$, where in velocity control this translates to the joint velocity in rad/s, and with position control to the target position in radians. With velocity control the clamp action could be achieved by choosing only positive or negative velocities, as clamping could be achieved by bending a joint. If position control is used, the position to clamp has to be chosen from the action space to clamp. As multiple actions are taken within one second, a next action could change the direction the joint was going for position control. This means that actions with equivalent positions have to be taken after each other, where this is not the case for velocity control. In velocity control only the sign of the value matters. The reason continuous actions are chosen is because the joints still need to able to change with which speed the joint is changing. Since, actions can only be taken in discrete time, and sometimes small changes in joint position are required. The camera and the rotatable section are controlled by position control as with these joints it is not important that they learn to clamp. The action space is defined in Table 3.1.

Table 3.1: Joint and wheel control methods, with the corresponding action space.

	Joint1	Joint2	Joint3	Joint4	Joint5	Joint6	Wheels
Control Method	Vel	Vel	Pos	Vel	Vel	Pos	Vel
Space	$[\pi,-\pi]$	$[\pi,-\pi]$	$[\pi,-\pi]$	$[\pi,-\pi]$	$[\pi,-\pi]$	$[\pi,-\pi]$	(4,0,-4)

3.2.3 Observations

In the analysis several observations were discussed. To perceive the differences in performance the observations are divided into three groups. These observations will be used in the experiments which are explained in Chapter 4. The first group consists out of minimal observations. This group is called minimal observations. This group consist out of the observations shown in 3.1. Where q_j are the joint positions, p_w the wheel positions in absolute space, o_j the orientation of the joints in absolute space and A_{prev} the previous actions of the previous time step. The minimal observations is created to check if these observations are enough for reasonable performance. The location of the joint and wheel positions are illustrated in Fig. 3.2. Only two positions and orientations are given to see if the RL can distinguish sufficiently between states, with less information. In principle the front and back positions could give enough information

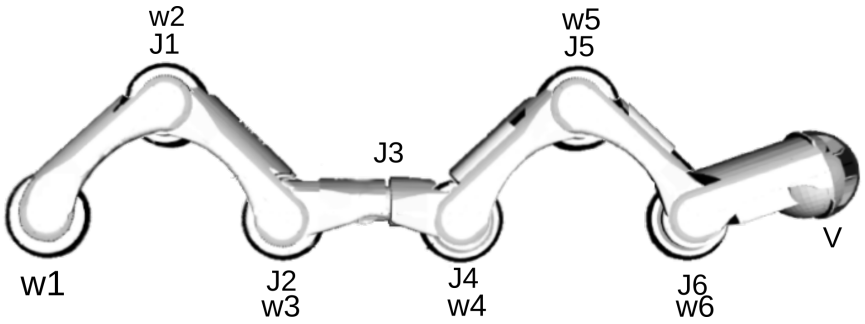


Figure 3.2: Schematic image of the PIRATE, containing the location numbers of the joints and wheels. Constructed from Dertien (2014).

whether it is straight or entering a turn. Furthermore, relations between the positions and joint positions could be made which might be sufficient to distinguish certain postures. Two orientations should provide enough information about the relation with the rotatable joint section and whether the PIRATE is driving horizontally or vertically. Furthermore, the wheel velocities are left out as based on the posture the agent should able to figure out when to apply a certain wheel velocities. Also the actions of the previous time step are added as observations. By performing preliminary experiments, this showed that adding these previous actions stabilizes the behavior of the PIRATE greatly.

$$\begin{aligned} q_j &= [q_{j1}, q_{j2}, q_{j3}, q_{j4}, q_{j5}, q_{j6}] \\ p_w &= [p_{w1}, p_{w6}] \\ o_j &= [o_{j1}, o_{j6}] \\ A_{prev} &= [A_{j1}, A_{j2}, A_{j3}, A_{j4}, A_{j5}, A_{j6} A_{w1}, A_{w2}, A_{w3}, A_{w4}, A_{w5}, A_6] \end{aligned} \tag{3.1}$$

The next group is called medium observations group. This group contains everything of the minimal observations, but now all wheel positions and orientations of the joints are added. Additionally, the velocities of the joints and wheels are added. Also the orientation and position of the vision sensor is added, which is in fact the end effector. In comparison to the minimal observations group, this group contains substantially more observations. This is shown in 3.2.

$$\begin{aligned} q_j &= [q_{j1}, q_{j2}, q_{j3}, q_{j4}, q_{j5}, q_{j6}] \\ v_w &= [v_{w1}, v_{w2}, v_{w3}, v_{w4}, v_{w5}, v_{w6}] \\ p_w &= [p_{w1}, p_{w2} p_{w3}, p_{w4} p_{w5}, p_{w6}] \\ o_j &= [o_{j1}, o_{j2} o_{j3}, o_{j4} o_{j5}, o_{j6}] \\ v_j &= [v_{j1}, v_{j2} v_{j3}, v_{j4} v_{j5}, v_{j6}] \\ A_{prev} &= [A_{j1}, A_{j2}, A_{j3}, A_{j4}, A_{j5}, A_{j6} A_{w1}, A_{w2}, A_{w3}, A_{w4}, A_{w5}, A_6] \\ V &= [V_p, V_o] \end{aligned} \tag{3.2}$$

The third group will consist out of medium observations but combined with vision observations. The group is called vision observations group. This will allow the PIRATE to see what is coming.

In the analysis some observations are mentioned which are not included in the final group observations. These are absolute joint velocities and torque. Although it would be interesting to see if these observations can improve the solution, there are left out as there was no time to research this properly. This can be something for future work.

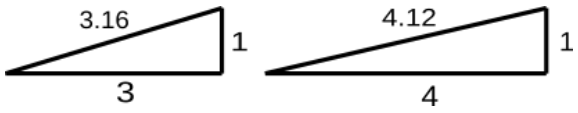


Figure 3.3: Illustration to explain that a skewed distance is less beneficial.

3.2.4 Main Reward Function

A reward function often used in this thesis is described in this section. As this is the main reward function used within the thesis, this reward function is labeled as main-reward. The main goal of the PIRATE is to move forward in the pipe. To achieve this there has to be a reward function which rewards the PIRATE for going forward and or punishes it for going backward. As mentioned earlier, a lot of approaches do this by defining a target location and give a reward based on the distance of this target. This is not feasible if the pipe system has several bends, because if the end location might be close to the starting location. In that case the agent has to discover that it could potentially receive more reward by first going farther away. As this slows down the learning, another approach is taken. The main-reward is based on measuring a distance in a previous time step and comparing this distance to the current time step. To elaborate upon this, it measures the distance in time step 1 from bending joint 2 to bending joint 4. Also, it saves the position of bending joint 2. In the next time step, the distance is measured from the saved position of bending joint 2 to the current location of bending joint 4. The joint positions are illustrated in Fig. 3.2. Then both distances measured in the different time steps are subtracted. If a scalar positive is obtained, the PIRATE went forward. If it is negative the PIRATE went backward. Also, the PIRATE receives more rewards for larger distances, this motivates it to move faster in the pipe system. In Eq. (3.3) is shown mathematically, where d is a function which calculates the Euclidean distance between two points. In Fig. 3.4 this principle is illustrated.

The reason to measure the distance from joint 2 to joint 4 is that the robot is not able to change this length. Since joint 3 is rotatable joint, it cannot influence the distance between joint 2 and 4. The problem that could arise if these lengths can change is that it would influence the reward. If for example, the reward is based on the length between joint 2 and joint 5 the reward could be affected by the angle of joint 4. If the angle of joint 4 is at 180 degrees the distance between joint 3 and 5 is larger than if joint 4 has a smaller angle like 45 degrees. Therefore, if joint 4 is at a 180 degrees angle, the PIRATE will receive a negative reward if it then clamps joint 4. However, clamping joint 4 might be vital in order to properly drive forward.

Between the measurement of the two time steps, joint 2 could change its angle. This would effectively make the measured distance longer or shorter, than was actually driven. This might introduce unwanted behavior, but this is probably not the case. When it is changing the angle it could actually receive a little bit more reward. However, constantly changing its angle to receive more reward would simply interfere with the goal to drive forward. Something that should be considered is what happens when the rotatable section is skewed, when driving forward. If the rotatable section is skewed the PIRATE will receive less reward. This is explained with Fig. 3.3. If the PIRATE would drive forward with a skewed rotatable section the reward $4.12 - 3.16 - = 0.96$ would be less then keeping the section horizontal $4 - 3 = 1$. The PIRATE would thus receive more reward by keeping its section straight.

Something that should be taken into account is that if the pipe system has a vertical drop, this reward function stimulates the PIRATE to let itself fall. This could be easily prevented by implementing a maximum distance per time step. If this is exceeded a negative reward could be given if this maximum is exceeded.

A possible improvement that can be made to the reward function is to increase the reward exponentially by the distance. This could stimulate the PIRATE to drive long distances. However,

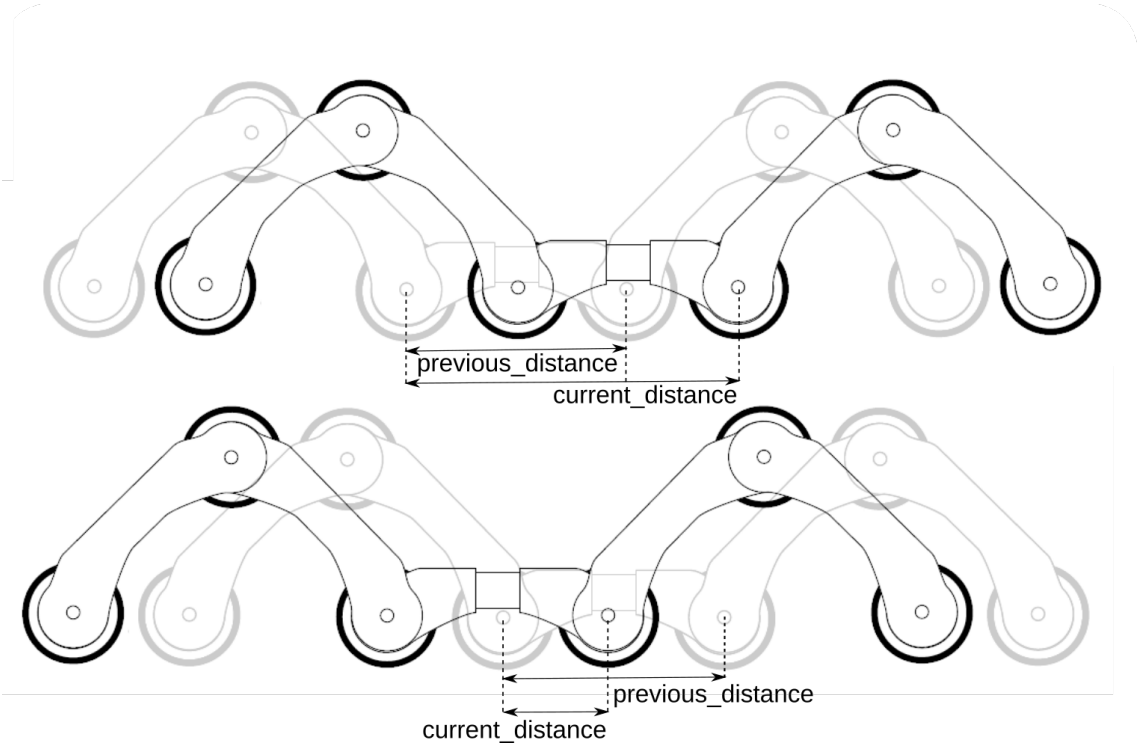


Figure 3.4: Illustration the distances used to compute the main-reward. Gray: PIRATE is in time step t-1. Black: PIRATE is in time step t. The image is constructed from (Dertien, 2014).

the reason to not increase the reward exponentially over the distance is that this makes it easier to add other reward functions. Adding more reward functions might be necessary to correct unwanted behavior. If the main reward becomes very large, other rewards might be insignificant in comparison.

This reward function does not additionally punish the PIRATE for not moving forward. Giving a negative reward each time step could encourage the PIRATE to complete the episode faster. However, as future rewards are discounted by gamma, the PIRATE will still be encouraged to take actions which will complete the episode as fast as possible. This makes adding a negative reward each time step redundant.

$$R_1 = d\left(p_{j3,(t-1)}, p_{j4,(t)}\right) - d\left(p_{j3,(t-1)}, p_{j4,(t-1)}\right) \tag{3.3}$$

$$= \text{current\_distance} - \text{previous\_distance} \tag{3.4}$$

3.2.5 Training Considerations

To properly train the PIRATE using a specific RL algorithm, some careful considerations have to be made. Important aspects are the choice of hyperparameters, training duration and training reproducibility. In this section these subjects are outlined. The final design choices follow from this analysis.

RL algorithms have several important hyperparameters (AurelianTactics, 2018). For example, the hyperparameters of PPO include minibatch size, entropy coefficient, clip, gamma, lambda, kl target. Most parameters have a specific range wherein the algorithms are likely to perform reasonably. Tuning the hyperparameters might be more important than which specific RL algorithm is used. However, optimizing these parameters is time consuming and done by trial and error. There are multiple ways to tackle this problem. Two common concept approaches are sequential optimization and grid search. In sequential optimization the agent performance

is evaluated after a training is completed. The next experiment then slightly alters the hyperparameters to possibly improve performance. The hyperparameters are compared with the corresponding performance and new hyperparameters are chosen. However, there are numerous combinations of hyperparameters that are examined, as the hyperparameters are adjusted after each training to search for a possible improvement. Therefore, sequential optimization is rather slow. In grid search several experiments are performed in parallel. This is faster than sequential optimization, but requires better computational hardware. To free up hardware resources, underperforming experiments could be prematurely stopped. A new experiment could then be started such that hardware resources are not wasted. This concept of stopping experiments early is implemented in Hyberband (Li et al., 2018) and Vizier (Golovin et al., 2017). Another method to efficiently tune hyperparameters is by using a population based training algorithm (Jaderberg et al., 2017). Population based training runs parallel sessions similar to grid search. The main difference is that for population based training the policy of the best running agent is transferred to all the other agents after a certain time interval. Therefore, underperforming experiments do not start over, but immediately benefit from a good performing agent. After the transfer, the hyperparameters of the individual experiments are changed to search for a better solution. Generally these new hyperparameters are based on the hyperparameters of the best performing agent. As hyperparameters can change halfway through training, using this approach even better results could be found. The agent could perform better with other hyperparameters, because the agent interacts differently with the environment then at the beginning of the training. This is simply because it acquired new skills which could drastically change the experience in the environment. Due to this, changing the hyperparameters could optimize the current state of learning in a better way. A simple example for changing hyperparameters halfway the experiment is lowering the learning rate hyperparameter when the agent starts to converge. Applying smaller updates to the policy could prevent bouncing around the optimal solution.

However, the experiments are not performed with population based training as it also has some disadvantages. The disadvantage is that population based training does not aim at finding good values for the hyperparameters. During population based training the hyperparameters are frequently changed in order to optimize the final solution. However, as the hyperparameters are changed during training it becomes quite hard to determine the final values for the hyperparameters. This means that all experiments should be performed with population based training. Performing all experiments with population based training with the current hardware would take to much time. Therefore, population based training actually is not suitable for this experiment. For this reason grid search is preferred to population based training.

The next training consideration concerns the stochastic action distribution of PPO. As PPO uses a stochastic action distribution, the actions the agent performs might differ, even if the model and the inputs are equal. This ensures exploration, which is explained in Chapter 2. However, as actions taken by the agent will differ for every experiment, experiment performance will diverge as well. This could happen if the agent continuously performs actions which are low in reward. This might lead to a underperforming experiment. If multiple identical experiments are performed, an underperforming experiment can be recognized. Thus, to ensure properly trained agents, the experiments have to be done repeatedly. As the underperforming experiments can be identified, this ensures that results from different experiments can be compared. However, performing multiple experiments will significantly increase training time. To speed up training, poor performing experiments can be stopped prematurely. Prematurely stopping is the similar approach that is used to find hyperparameters faster.

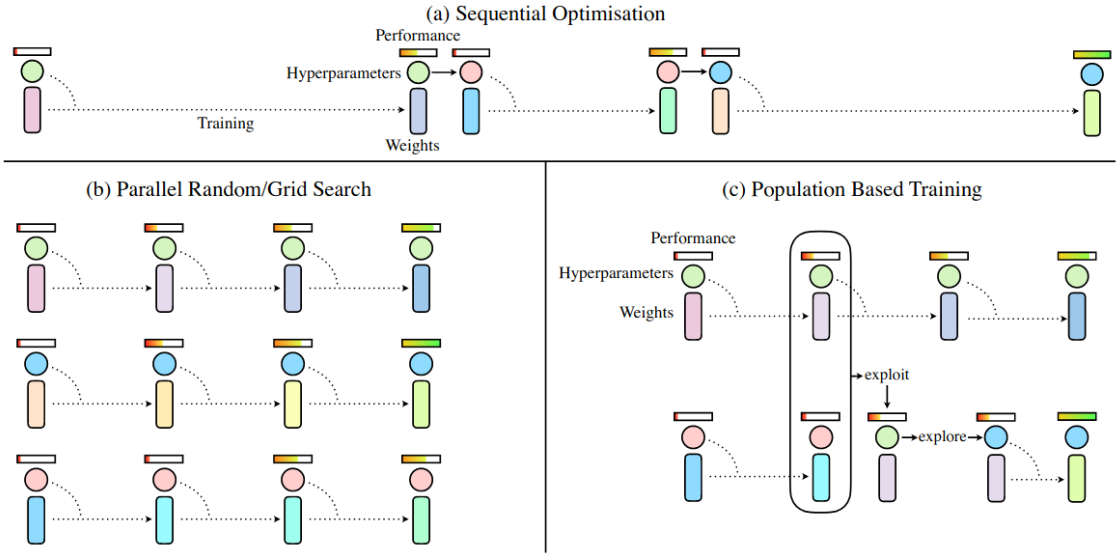


Figure 3.5: Population based training, (Jaderberg et al., 2017).

3.3 Hierarchical Reinforcement Learning

After several experiments it became clear that certain aspects in the dynamic environment are too difficult to learn with the proposed setup, which is described in Section 5.3.3. The results and problems of the static and dynamic experiments are discussed and presented in Chapter 5. As an attempt to solve this, Hierarchical Reinforcement Learning (HRL) is implemented. The reason HRL could solve the problems described in Chapter 5 is that specific policies can be defined to solve specific sub-tasks. The main problem of the current RL approach is that it is not able to take a turn or leave a turn. The suspected reason for this is that learning to drive interferes with learning to make a turn. Also, the sparse rewards make learning difficult. A sub-task which is specialized in taking a turn or leaving a turn could solve these problems. As these sub-policies will be specialized in solving certain problems, they could contain there own inner reward functions. First, the suggested overall setup is discussed in Section 3.3.1. After this setup is introduced, the corresponding observations and reward functions are discussed in Sections 3.3.2 and 3.3.3. The analysis and methods of this section are based on previous results.

3.3.1 Overall Setup

The suggested setup exists out of three sub-policies and one master policy. One sub-policy is dedicated to driving through straight pipes. This policy is named the clamp-drive policy. The other two sub-policies, named enter-turn policy and outward-turn policy are a specialized in entering and leaving a turn. The reason this exist out of two policies, is that entering and leaving a turn are still complicated tasks. Also, defining a reward function specialized for taking an entire turn is difficult. Splitting this into two separate sub-policies simplifies the problem to define suitable reward functions. The enter-turn policy is trained to enter the turn with the front part. The outward-turn policy then is defined to be able to leave the turn. The master policy decides which sub-policy is active in the environment. A schematic representation is shown in Fig. 3.6. Specifics of the policies are explained in the next sections. The exact design and implementation is explained in Section 4.5.

There are of course multiple sub-policy combinations. Some possible sub-policies are discussed, and why these other models are less preferred. For example, it is possible to split the action space of the joints and wheels into two separate sub-policies. The advantage of this is

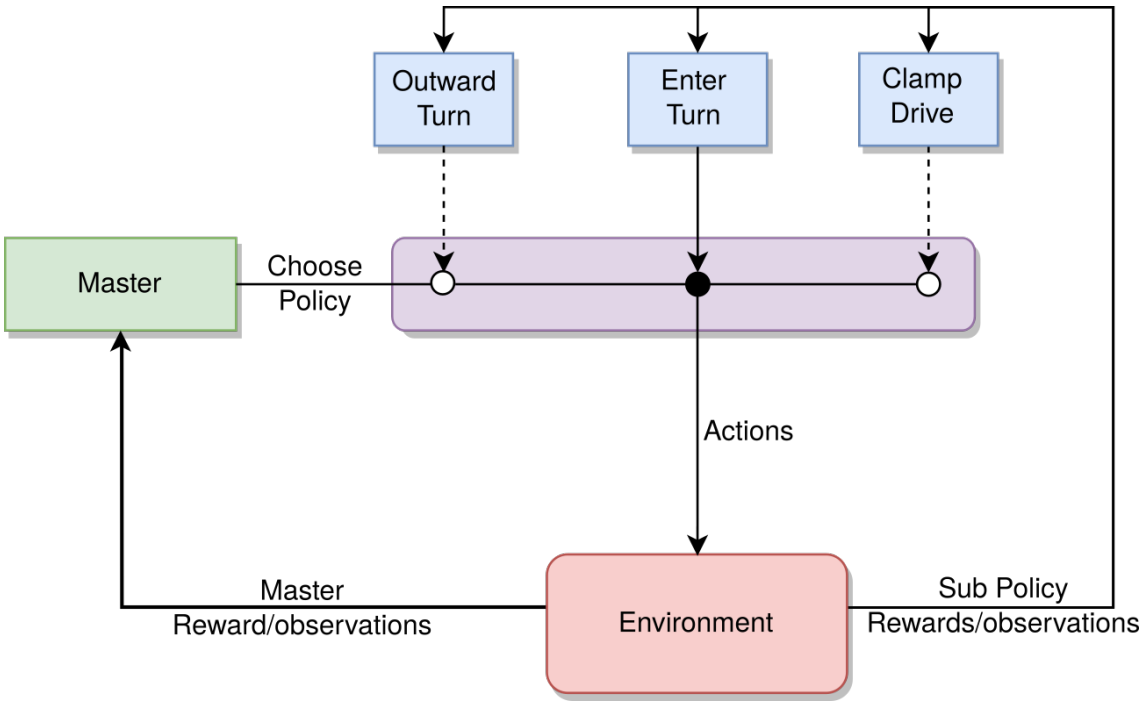


Figure 3.6: Schematic representation of the hierarchical setup.

that a sub-policy either has to learn to clamp or to drive. This reduces the amount of possible combinations of actions and makes the reward functions more straightforward. The disadvantage of this is that the driving through a pipe is highly dependent on the joint configuration. If the the joints are not lined up correctly, driving will be very difficult. As the goal of the clamping sub-policy is only to clamp, it will not try to clamp in such a way that driving will become easier. Therefore, this idea is not pursued.

Another option would be to define sub-policies to split the control of the PIRATE in half. One policy could control the front including the rotatable section, and another policy could control the back. As one policy only has to control the front section, it can focus on entering a turn. The problem is that to enter the turn the front section should move forward. As the front section tries to maneuver itself through, the backside has to take care of driving forward. Thus, these sub-policies will result in the same problems as with the clamp and drive sub-policies. As one policy depends on the other, on which it has no control, this could form problems.

When sub-policies depend on other policies complications can arise. This could be solved by adding more hierarchy. For example, a sub-policy which can control subsub-policies and knows the overall goal of these subsub-policies it controls. The disadvantage of this is that the overall setup becomes much more complicated then the original suggestion. Furthermore, the advantages are probably minimal, compared to the suggested setup.

Clamp-Drive Policy

The first sub-policy is called the clamp-drive policy. The reason for this is that this sub-policy should handle the driving and proper clamping within a straight pipe. The PIRATE will be given rewards for going forward, with the main-reward. Additionally, rewards will be given to the PIRATE to clamp itself. As the main-reward does not specify how it should clamp, sometimes the solutions are sub-optimal. Adding a clamp-reward will ensure that the clamping capabilities are fully used. This can be done because the sub-policy is not indented to move through a turn anymore. The same action space is used as in the SRL experiments, only now the clamp direction is defined by preventing positive velocities on Joint1 and Joint5. The clamp direction is

Table 3.2: Action space of enter-turn policy.

	Joint1	Joint2	Joint3	Joint4	Joint5	Joint6	Wheel0	Wheel1	Wheel2	Wheel3	Wheel4	Wheel5
Control Method	Vel	Pos	Pos	Pos	Pos	Pos	Vel	Vel	Vel	Vel	Vel	Vel
Space	[0,-π]	[π,-π]	[π,-π]	[π,-π]	[π,-π]	[π,-π]	(4,0,-4)	(4,0,-4)	(4,0,-4)	0	0	0

defined to align the clamping direction with the other sub policies. If the other sub-policies will clamp in a different way the PIRATE could fall by switching policies in a vertical pipe section.

Enter-Turn Policy

The main goal of the enter-turn policy should be entering, or at least orienting itself properly for a turn. The reason orienting could be enough is that if the PIRATE is oriented properly, the clamp-drive policy can achieve entering as well. This is based on the results explained in Section 5.3.1. To design an efficient turning policy, one particular thing should be avoided. In the dynamic experiments shown in Section 5.3.1 the PIRATE was unable to make a turn because it was totally clamped. When it is totally clamped it is unable to rotate itself. When designing the enter-turn policy it is thus particularly important that the front part does not clamp. To accomplish this, several things were attempted. A reward to stretch could promote this behavior. A disadvantage of this, is that the reward function does get more complicated. Due to this the agent struggles with learning. Also the main-reward has to compete with the stretch-reward. If the stretch-reward is much bigger then the reward for going forward, going forward is not that important. Another huge problem is that the stretch-reward will become less when the PIRATE is entering a turn, because in the turn the PIRATE has to bend its joints to perform a turn maneuver. Thus, it does not make sense to use a stretch-reward, as it will discourage the PIRATE to make the turn. This might be solved by making the main-reward more important, compared to the stretch reward. However, this might stimulate to clamp again, as this makes driving forward easier.

Due to these contradictions a different approach seems necessary. Disabling the front wheels, makes clamping the front side not such an attractive behavior as it cannot use the front wheels anymore to drive forward. The action space is defined in Table 3.2. If the wheels are disabled, it would complicate driving forward when clamped. Of course the PIRATE should still be able to go forward with the three active wheels at backside. Based on results discussed in 5 from experiments without a hierarchical structure this looks feasible. Also, as the enter-turn policy is only used for entering the turn, therefore not being able to drive properly actually makes sense for the master policy. This will be explained in the upcoming section, which explains the master policy. To stimulate the clamping of the backpart of the PIRATE, a clamp-reward function is applied. The direction is defined by only allowing negative velocities. If the backpart of the PIRATE is clamped, it cannot leave the turn if it is not properly oriented. This is when the outward-turn policy should take over.

Outward-Turn Policy

The third sub-policy is the outward-turn policy which should continue where enter-turn policy terminated. Which makes the task of the outward-turn policy to leave the turn. As the task for leaving the turn is precisely the opposite of the enter-turn policy, it seems logical that the sub-policy should be configured as the exact opposite of enter-turn policy. In the outward-turn policy the wheels in the back part of the PIRATE are disabled. Again a clamp-reward is applied to the front section. The action space is defined in Table 3.3.

Master Policy

The master policy determines which sub-policy can determine the current actions. The master policy only takes an action after a sub-policy is finished. This action thus only determines the

Table 3.3: Action space of outward-turn policy.

	Joint1	Joint2	Joint3	Joint4	Joint5	Joint6	Wheel0	Wheel1	Wheel2	Wheel3	Wheel4	Wheel5
Control Method	Pos	Pos	Pos	Pos	Vel	Pos	Vel	Vel	Vel	Vel	Vel	Vel
Space	$[\pi,-\pi]$	$[\pi,-\pi]$	$[\pi,-\pi]$	$[\pi,-\pi]$	$[0,-\pi]$	$[\pi,-\pi]$	0	0	0	(4,0,-4)	(4,0,-4)	(4,0,-4)

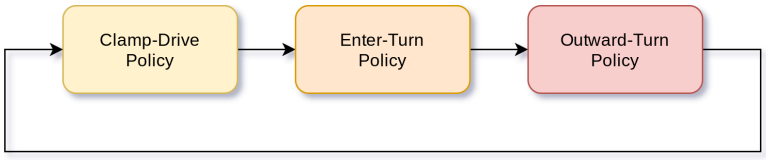


Figure 3.7: Ideal sequence of the master policy.

next sub-policy. Therefore, the master policy only receives observations of environment when a sub-policy ends. The master policy is located on a different timescale, than the sub-policies. This is possible due to the SMDP. It is important that the master policy is able to distinguish between different situations in which it should apply a certain sub-policy. As the master policy only has one time slot to choose a policy, the camera should point to an interesting direction. If it is for example pointed at the ceiling, the master policy is not able to see an upcoming turn. If the master policy then chooses the wrong sub-policy it might become stuck. This could also happen if the sub-policies take to long and the master policy receives sparse observations. Because the master policy operates on slower timescale, it could take longer to train. What could help is that the master policy probably operates in a sequence. There will be a constant sequence of driving to a corner, entering a bend and leaving a bend. This is shown in Fig. 3.7. An LSTM could exploit this possible sequence. Depending on how the sub-policies are precisely implemented the sequence could differ.

The goal of the master policy is to let the PIRATE travel through the pipe system. Considering this, the main-reward mentioned before can be used. Although the rewards of the master policy are probably considerably larger then the sub-policies this in principle should not matter, because the optimizer tries to minimize the loss. However, since master policy actions take a longer time, mapping a correct value function with larger rewards might be more difficult.

Solutions to ensure that the master policy makes a proper decision, could be done by allowing the master policy to make multiple decisions based on multiple observations, which then determine the final choice. These decisions could happen before the sub-policy actually ends.

3.3.2 Reward functions

In the sub-policies several extra rewards are added to stimulate certain behavior. However, learning with these rewards is more complex, as the agent has to figure out which action corresponds to which part of the reward. This is especially difficult as some rewards will contradict each other.

Clamp-Reward

The intention of the clamp-reward is to let the PIRATE clamp. This is done by setting a negative reward between the distance of the wheels that need to be clamped, which is shown in Fig. 3.8. It might be a good idea to exponentially increase this value such that there is a considerable larger reward difference between unclamped and clamped. Since, the distance between a clamped and unclamped state is not that large. When figuring out how to clamp, a larger difference in the advantage will be present.

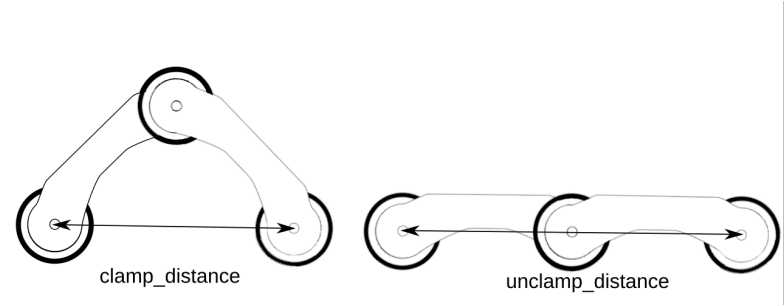


Figure 3.8: Clamp reward, image constructed from (Dertien, 2014).

Stretch-Reward

The stretch-reward is based on the same principle as the clamp reward. The difference is that the reward is now positive instead of negative. This will ensure that the PIRATE will stretch itself.

Depth-Reward

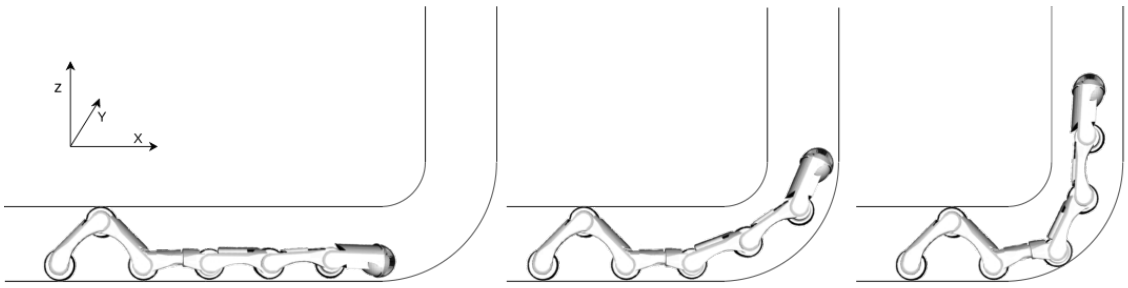
When the PIRATE starts to learn to drive forward, the camera observations are not particular important. However, the camera section could be helpful to clamp itself better in the pipe system. But when it is using the camera section to clamp, it is not aware of the upcoming corner. The purpose of the depth-reward is preventing that the camera section points to the sides of the pipe. As the camera is a depth camera, depth information could be used to construct a reward function. Of the depth information the smallest distance is taken as the reward. If the camera starts to point more to the side of the pipe, the distance will decrease. To obtain the highest reward the depth camera has to point towards the middle. The reasons the minimum distance is used and not the largest distance, is that this could contradict with the main-reward. If the PIRATE, for example drives to a corner this distance will decrease as well. Using the minimum distance this will not have such an effect.

3.3.3 Sub-policy Observations

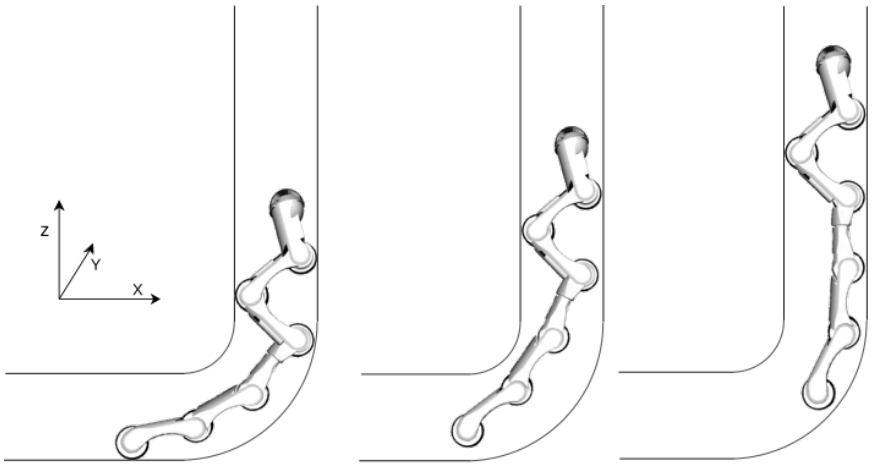
Determining the proper observations for the sub-policies could be based on results from the dynamic and static experiments. In these experiments it turned out that the RL with medium observations often outperforms minimal observations. Based on these experiments is assumed that all sub-policies will benefit from these observations as well. The options that remain based on the previous experiments, are adding camera observations and an LSTM layer. Below all sub-policies are briefly discussed whether they could benefit from adding camera observations or an LSTM layer.

The clamp-drive policy only drives through straight pipe sections. Therefore, the camera observations seem unnecessary. However, the problem is that when the clamp-drive policy ends, the master policy does need the camera observations. If the clamp-drive policy does not take into account that the camera should point through the pipe, instead of to the ceiling, the master policy is not able to make a proper decision. Due to this the camera observations will be included and a depth-reward is added. This ensures that the clamp-drive policy will try to point through the pipe. This problem applies applies to all sub-policies.

The enter-turn policy probably has the most difficult task. It certainly needs the camera observations to maneuver itself through the pipe. Implementing an LSTM could be helpful as entering a pipe probably requires performing a sequence of actions. Also it can help to remember in which direction the bend was. Experiments should be performed to see the effect of this. By performing preliminary experiments it became clear that adding a depth-reward has a



(a) Enter turn policy schematic



(b) Outward turn policy schematic

Figure 3.9: Feasibility Results

negative effect on the performance of the enter-turn policy. This is probably because the enter-turn policy mainly controls the front section to enter the turn, adding a reward based on this movement makes the reward function more complex. Because the aim of the enter-turn policy it to enter the turn properly, the depth-reward is also redundant.

The outward-turn policy probably also doesn't need the camera observations as it cannot see what is happening in the turn. However, the camera observations are included because of the same reasons as in clamp-drive policy. Because of this, the depth-reward is added as well. Also performing this policy is less difficult as entering a turn. It is less difficult as it is much harder to get stuck when leaving the bend. An LSTM will because of this probably not have much effect.

As analyzed there might be a benefit of using an LSTM within a sub-policy. However, this might cause some problems when used in the HRL. As the sub-policies will perform a certain amount of time, which is specified by the master policy, the LSTM might receive observations which are not actually within the same sequence anymore. Or it simply has to start over. Because a sub-policy can end, observations are not lining up correctly as was the case when training individually. Knowing this, the LSTM solution might not perform so well within a HRL environment and should be avoided.

3.4 Conclusion

In this chapter the analysis and several methods are introduced. This section outlines the most important concepts of this chapter. Learning a good RL solution will be challenging due to several reasons. One of these reasons is the complicated continuous action space of the PIRATE

existing out twelve controllable joints. With this action space some difficult maneuvers such as entering and leaving a turn should be performed. Something that makes this even more complicated is that the pipe system has sparse features and looks similar.

The main-reward function is based on the distance driven. This is required because the Euclidean distance between the target en the PIRATE might have to increase first before the PIRATE is able to reach the target.

To see performance differences between different observations spaces, 3 groups with different observations are composed. The first group, minimal observations, contains observations that are considered to be crucial. The next group, medium observations, contains more observations which seem useful for better performance. The last group, vision observations contains the observations of the medium observations with the addition of depth images. The RL algorithm used in this thesis is Proximal Policy Optimization (PPO). PPO is chosen because it seemed to perform better in unstable environments. For a successful RL solution, the hyperparameters specified for training are very important. To find these proper hyperparameters grid searches can be performed.

After performing several experiments based on a single RL solution. A hierarchical setup is proposed. This setup contains three sub-policies. The clamp-drive policy is intended for driving through straight pipes. Furthermore the enter-turn policy is designed to enter turns, where the outward-turn policy should leave the turn. These sub-policies are then controlled by one master policy. All policies have their own rewards, observations and actions.

4 Design and Implementation

In this chapter the design and implementation are presented. In Section 4.1 the general design implementation is explained. This includes the explanation about how the PIRATE is implemented in V-REP. Afterwards, in Section 4.2 the design of the feasibility experiment is explained. In the feasibility experiments a discrete action space is used to simplify the experiments. After the feasibility, Section 4.3 explains which methods are used to search for optimal hyperparameters. In the hyperparameter experiments the continuous action space specified in Chapter 3 is used. The first implementations of the RL experiments are described in Section 4.4. Finally in Section 4.5 the hierarchical design and experiments are presented.

4.1 General Design Implementation

4.1.1 Tool Analysis

Using a RL framework could help speed up the progress in this research. As machine learning steadily becomes more popular, also the amount of different tools is growing. Picking the right tool could save a lot of time. Therefore an exploration is performed. The tools are rated on several criteria which are chosen based on the analysis. For each criteria the tool can receive $[--,-,-/+,+,++]$ a certain amount of points, which translates to a 5 point score system with a range from -2 to 2. Each criteria also receives are certain weight to mark its importance. Most criteria have a weight of one. Criteria which receive a weight of 2 are: Multi Agent and Documentation. To realize a Hierarchical Reinforcement Learning (HRL) implementation the support for multi agent implementation is important. A multi agent implementation can support multiple agents in the same environment. Documentation is also listed as a high importance criteria as learning a new framework without proper documentation could be time consuming. As the amount of tools are quite extensive only four promising frameworks are treated. In Table 4.1 the results of this exploration are shown. The conclusion is that the framework Ray almost always has a better or equal score on every criteria.

4.1.2 Design implementation

In this section the implementation of the RL experiments are explained. For the implementation of the experiments a combination of the simulator V-REP (Rohmer et al., 2013) and the Ray framework (Moritz et al., 2018) is used. As stated in Chapter 2, Ray contains the RLlib library. OpenAI Gym support is available in RLlib and this is used to build the RL environment. OpenAI Gym provides a framework to build a custom RL environment (Brockman et al., 2016). In

Table 4.1: Assessment of the RL frameworks.

	Stable-Baselines	Baselines	Ray	Keras-RL
PPO	++	++	++	++
Multi Agent	-	-	++	-
Parameter Search	-	-	++	-
Keras	+	-	++	++
Tensorboard	++	+	++	+
Customizable	+	–	++	++
Documentation	-/+	-	+	+
Simplicity	+	–	+	++
Base Implementation	++	+	++	+
Score	7	-2	16	9

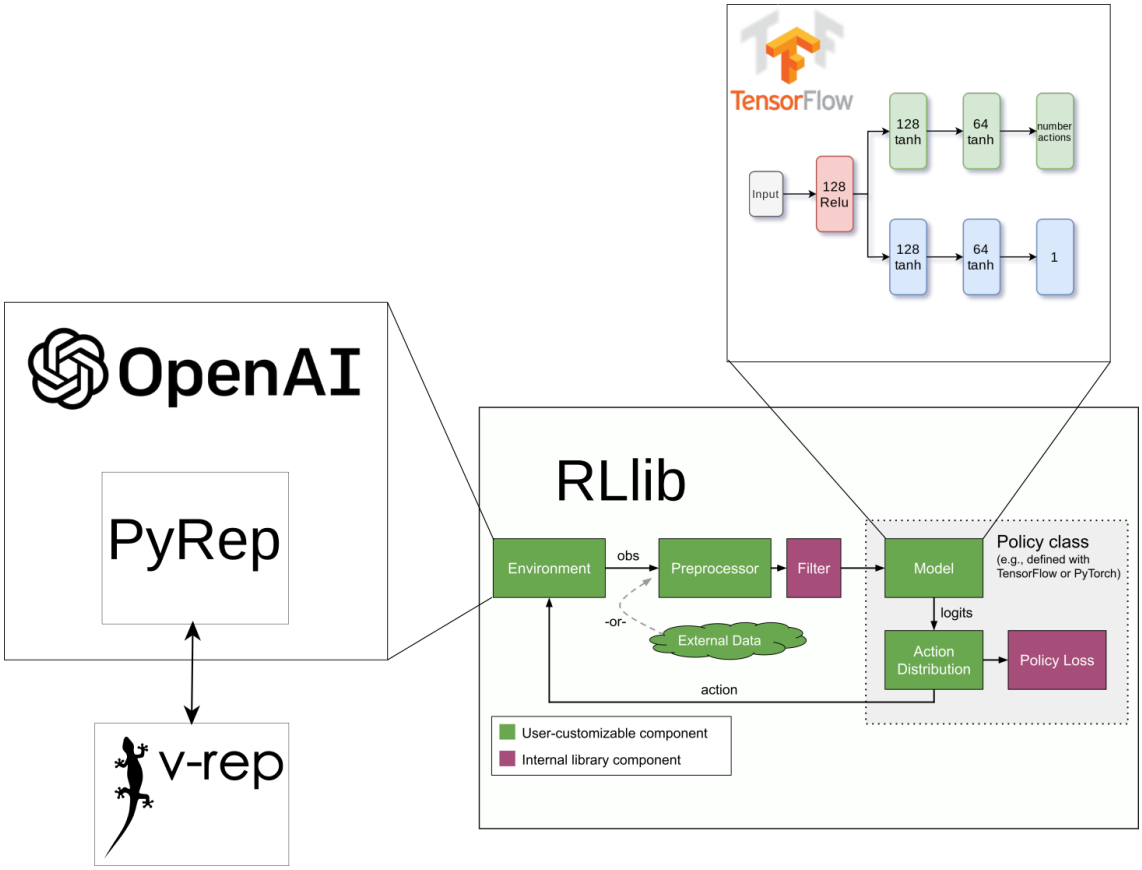


Figure 4.1: Schematic of implemented tools within RLlib, based on Anyscale (2020).

this thesis, V-REP is used as the simulation platform in the custom environment OpenAI Gym. RLlib communicates with V-REP by using PyRep. PyRep is used because the integrated Python API of V-REP is much slower (James et al., 2019). A schematic representation of these relations is shown in Fig. 4.1. This figure also shows which user-customizable components are implemented. Furthermore, RLlib is able to use the TensorFlow library, which is used to build the neural networks. In RLlib the PPO algorithm is used, this is achieved by using the PPOTrainer[1]. As mentioned in Chapter 2, RLlib contains trainer classes which contain the RL algorithms.

A process diagram of the implementation is shown in Fig. 4.2. This is explained as follows. Starting at V-REP, the simulator runs for 50ms and then pauses. The PyRep interface retrieves state observations of the current simulation. This data is processed such that it can be send to the RLlib agent. The RLlib agent uses this data to train and predict the action that should be taken next. Again, the PyRep interface with V-REP is used to send the predicted actions to V-REP. Then the simulator V-REP runs again for 50ms and pauses the simulation.

4.1.3 V-REP implementation

The development of the PIRATE model in V-REP is based on tutorials and design considerations of V-REP (CoppeliaRobotics, 2019b,a). Each section of the PIRATE exists out of a visual and dynamic part. The dynamic part serves as the parent of the visual object. In the current simulation it mostly serves as the visual representation of the model. The invisible dynamic model is used for dynamic interactions, with other dynamic models in the environment, such as the pipe system. To make a stable and fast model, the amount of triangles of the CAD model of the PIRATE that is provided is reduced. This is especially important for the underlying dy-

[1]The PPO implementation of RLlib can be found here.

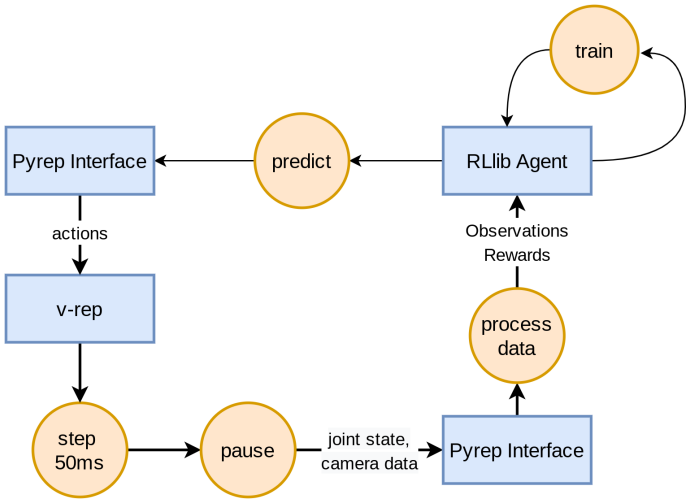


Figure 4.2: Process flow within one time step.

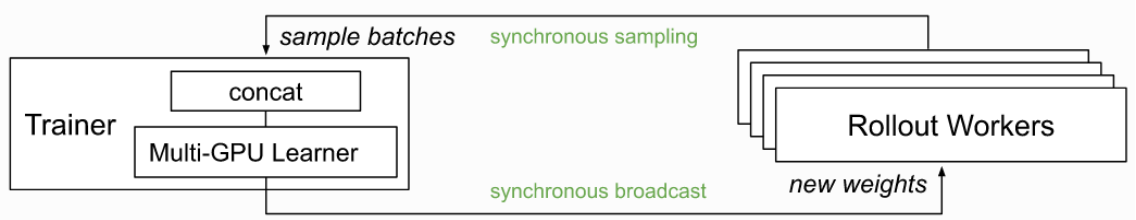


Figure 4.3: schematic of the PPO implementaiton in RLlib (Anyscale, 2020).

namic model. These detailed parts of the CAD data are morphed into simpler convex shapes. The convex shapes are symmetric and this thus means that the asymmetric shape of the PIRATE is not considered. On the visual model the amount of triangles is reduced as well, but as this model is not used for dynamic calculations within the simulation, a considerable amount of detail can be retained. The visual and dynamic V-REP model are shown in Fig. 4.5. As the visual part is mostly preserved it can be used, among other things, for more precise distance calculations between objects.

The V-REP model is build by connecting several sections together as a chain. All different sections are connected by joints. This can be seen in Fig. 4.4. These joints are defined as the bending_joints. Between each section a wheel is connected by a joint as well. However, as objects can only connect to one other object at a time, the wheels can only physically connect to one section of the PIRATE. This makes the implementation of the model different then its physical counterpart. If the wheels by its joints are actuated, this provides a force between the connected section and the wheel. In the actual PIRATE this force is probably divided over all sections, as the different sections of the PIRATE can lock each other physically. If the PIRATE becomes stuck, the wheel joint can push up, or push down a section. A bending joint is then affected by the force the joints of the wheels produce.

As already mentioned in Chapter 3, the focus of this thesis it not to build a realistic model of the PIRATE. The focus lies on building a stable model which could be used for RL. Especially as it is not straightforward to make a realistic model, which is also stable in the simulation. Dynamic model instability is highly dependent on the weight and inertia of the model objects. Model instability for example occurs if the relative weight difference between connected sections is to large. By trial and error it seemed that giving each part of the PIRATE a weight of 1 kg stabilized the model. After a bit of tweaking, increasing the weight of wheels a bit more seemed to have a stabilizing effect on the model as well. These weights are of course not realistic compared to

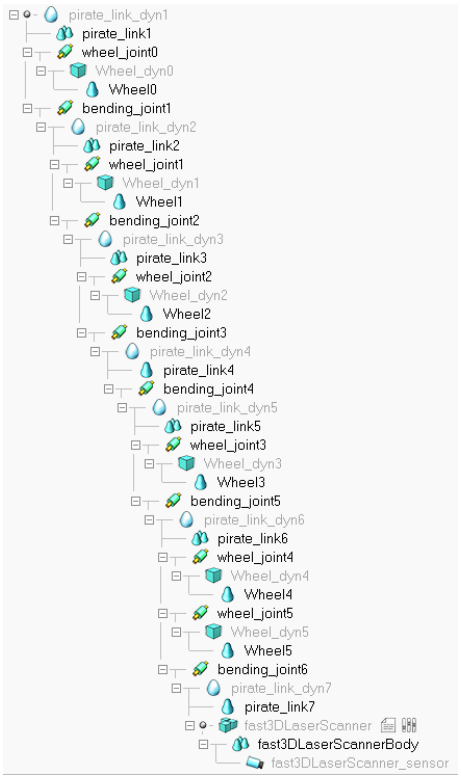


Figure 4.4: V-REP model structure of the PIRATE.

Table 4.2: Model configuration values used in V-REP

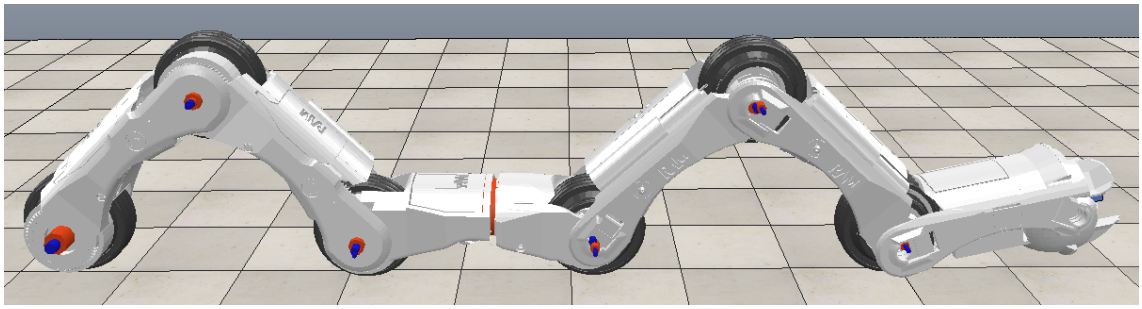
Properties	Inertia/mass (m^2)	Mass (kg)	Length (m)	Friction [2]	Torque (Nm)
Wheel	1.474e-02	1.2	0.45	2	50
Section	1.60e-02	1.0	1.22	0.71	50

the actual PIRATE. However, as a stable model outweighs the need for a realistic representation, some realistic design choices are not taken into account. This does not mean that a realistic model cannot be obtained, however significantly more effort should be devoted to optimize the model. This is simply not the aim of this thesis.

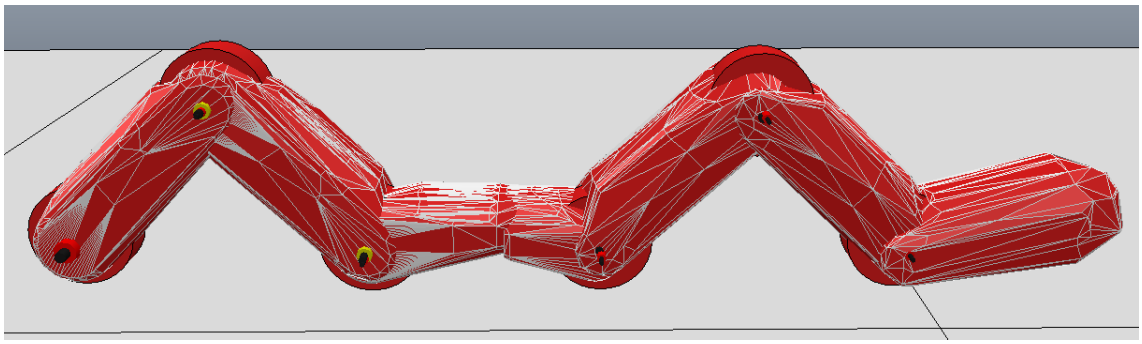
After a stable model was obtained, it is found that the dimension of the PIRATE in the simulation is quite large in comparison to the actual PIRATE. However, the size of the PIRATE is of course relative to the size of the pipe system. Due to time considerations, this is not changed. Future work could consider realistic values and sizes for the PIRATE. The most important model configuration values for reaching stability in the V-REP simulation are shown in Table 4.2. This values where obtained by performing the feasibility experiments explained in Section 4.2.

As adjacent sections of the PIRATE are physically overlapping, the PIRATE would become unstable if those sections are dynamically interacting. This is prevented by denying dynamic interactions between adjacent sections. This also means that adjacent sections could overlap much more then physically possible, which can be seen in Fig. A.1. To prevent this, the joint motors have a joint restriction. The motors are only able to turn 135° in each direction. Although, this is probably more then the physical PIRATE is capable of, large restrictions could slow learning solutions. Furthermore, the joint overlapping should not form such a substantial problem as the current pipe diameter is not large enough to allow such extensive rotations.

[2] This parameter is only intended for the Bullet physics engine.



(a) V-Rep Visual Model



(b) V-Rep Dynamic Model

Figure 4.5: V-REP models of the PIRATE.

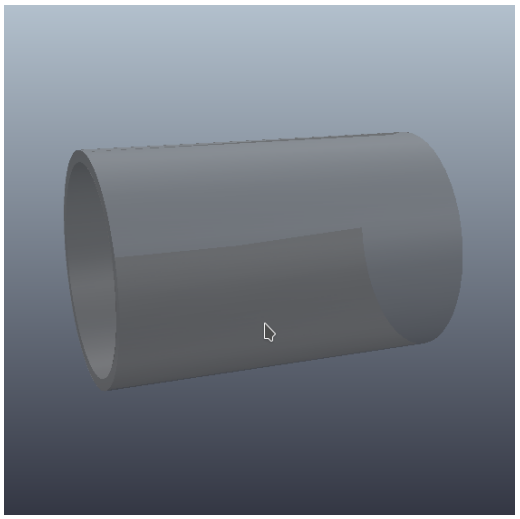
4.1.4 Simulation Environment

The pipe segments in V-REP are constructed from two CAD shapes. The first shape is a straight section shown in Fig. 4.6(a). The second shape is 90° corner shown in Fig. 4.6(b). These shapes are randomly put after each other such that a random pipe is build each time the environment is initiated. When a straight shape is placed, multiple straight sections are placed such that a longer straight section is created. The first pipe section could either start vertically or horizontally. To prevent that the PIRATE can leave the pipe system, the first pipe section is blocked with another shape. An example of a build pipe system is shown in Fig. 4.6(c). V-REP is able to support multiple physics engines. The physics engine used in this thesis is Bullet 2.83. This physics engine seems quite realistic, while at the same time delivering stable performance.

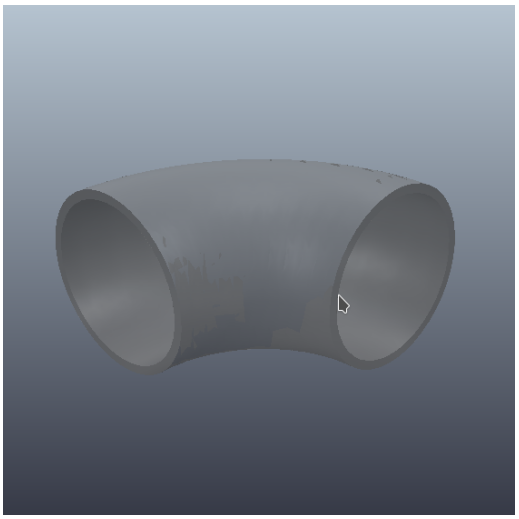
4.2 Feasibility Design

An essential experiment is to test the capabilities of the constructed PIRATE model in V-REP. Simple experiments are performed to determine whether or not the constructed simulation model is capable of driving through the simulation environment. After these experiments, experiments that determine the feasibility of the RL are performed. In these experiments simpler action spaces are used then specified in the Chapter 3.

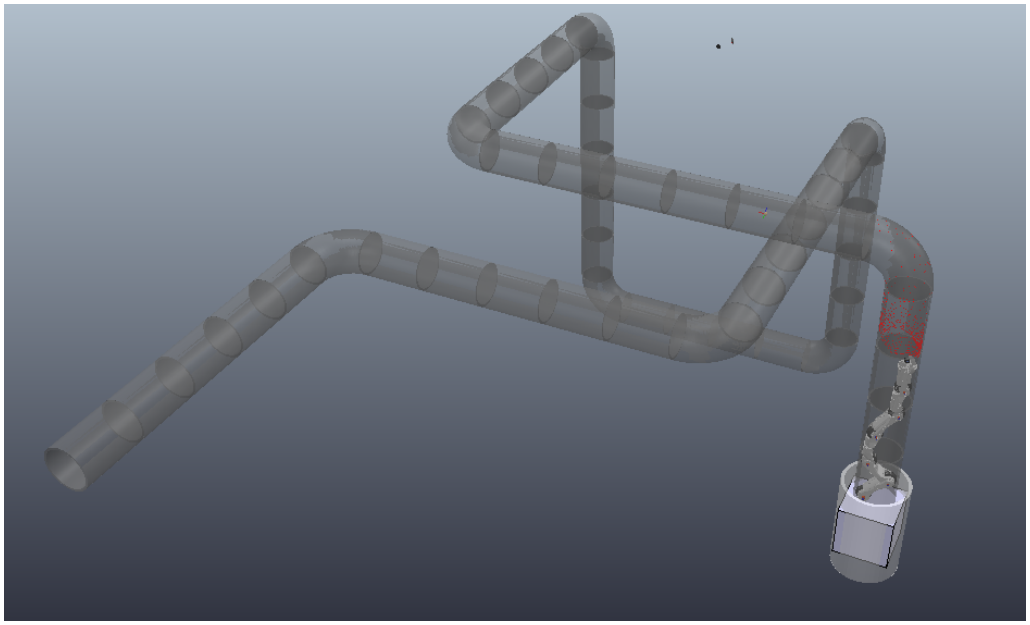
In the first experiment higher level actions are combined with a small GUI, such that the PIRATE could be manually controlled in the simulated environment. A high level action is, for example, performing a clamp on the backside of the PIRATE. In total eight high level actions are specified. They are given by: Clamp-Front, ClampR-Front, UnClamp-Front, Clamp-Back, ClampR-Back, UnClamp-Back, Reverse-Wheels and Pass. A clamp action means the PIRATE will clamp its joints. An unclamp action means no torque to the corresponding joints is applied anymore. Essentially the corresponding section is not actuated anymore. The ClampR function clamps in the opposite direction compared to the Clamp action. The ClampR function may be convenient when the PIRATE needs to go through a corner. Furthermore, the Reverse-Wheels



(a) Straight pipe section.



(b) Bend pipe section.



(c) V-Rep Dynamic Model

Figure 4.6: V-Rep pip system models.

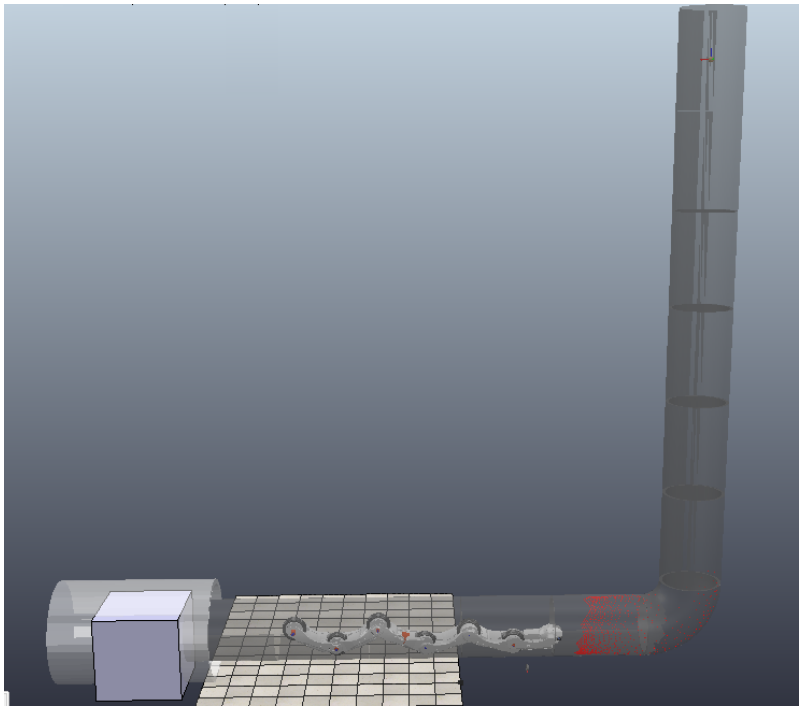


Figure 4.7: The V-REP feasibility environment.

action is able to reverse the velocity of all wheels, to allow the PIRATE to drive backwards. In the feasibility test there is no rotate action as the PIRATE is lined up correctly to enter the turn. Some of these actions could be runned simultaneously, where other actions will affect each other. For example, an unclamp or reversed clamp action neutralize a clamp action if they are applied to the same section of the PIRATE. Table 4.3 defines what the velocity of certain joints will be after executing a certain action. If the PIRATE is not clamped, all wheel velocities should be identical to drive forward. When the PIRATE clamps its front section, the middle wheel of the front section should reverse its velocity to maintain driving forward. As the wheel velocities are dependent on the configuration of joints, a reverse speed relative to the current speed of the wheel is applied, when a section is clamping. This is indicated with a minus sign, in Table 4.3. Thus, if the front section is clamped, the middle wheel should reverse its speed. However, if the action is applied twice, the wheel will reverse again. Therefore, the Pass action is defined to perform no new action.

The experiments are performed in an environment which contain a static pipe configuration. The pipe starts horizontally and consist of one 90° bend going up. This can be seen in Fig. 4.8. It is assumed that if the PIRATE is able to make a turn and drive straight up, it could also go down, or make a right or left turn. The reason to justify this assumption is that, when the PIRATE drives straight up through the pipe, it is vital that the PIRATE is properly clamped. In all other situations the task should be easier. Driving straight up requires the most effort as the PIRATE has to overcome gravity. If the PIRATE is not able to make a turn and drive straight up, the model should be adjusted, before continuing.

As the manual experiment succeeded, which is presented in Chapter 5, the RL feasibility experiment could be performed. Two experiments are performed to check the feasibility of the RL. The same environment is used as for the manual experiment, however now the episode ends when the PIRATE reaches the target placed within the environment. The environment is shown in Fig. 4.7. The difference between the two experiments is the action space. In the first experiment the higher level actions defined in Table 4.3 are used. The agent with the higher level actions will choose each time step one action. The OpenAI Gym Space is used for this is

Table 4.3: High level actions translated to joint and wheel velocities. Where the Joints J and Wheels W are defined as in Fig. 3.2.

Action Velocity	J1	J2	J3	J4	J5	W1	W2	W3	W4	W5	W6
Clamp-Front	2	-2					-				
UnClamp-Front	0	0					-				
CLampR-Front	-2	2					-				
Clamp-Back				2	-2					-	
UnClamp-Back				0	0					-	
CLampR-Back				-2	2					-	
Reverse-Wheels						-	-	-	-	-	-
Pass											

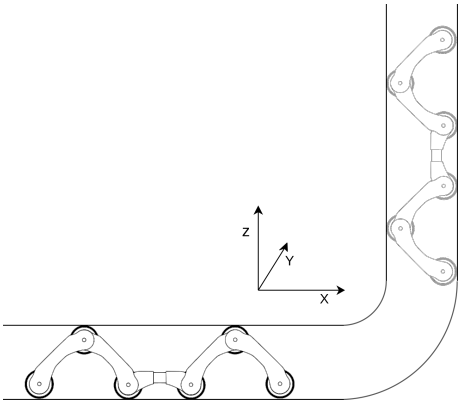


Figure 4.8: Schematic Feasibility Environment.

called: Discrete, shown in Eq. (4.1). The second experiment has a MultiDiscrete Space and allows the agent to choose the velocity of each joint and wheel separately in each time step. From now on these actions are defined as the lower level actions. For each wheel and joint there are three discrete actions possible, shown in Eq. (4.2). These three actions are translated by the agent to the three velocities: $[-2,0,2]$ rad/s. The lower level action space should thus be able to construct the same high level actions. The observation space used for both experiments are the minimal observations which were specified in Section 3.2. Furthermore, the main-reward for both experiments is used as the reward function. The corresponding neural network is in Fig. 4.11(a). The design of this neural network is explained in Section 4.4.1.

$$\text{High level action space: Discrete(8)} \tag{4.1}$$

$$\text{Low level action space: MultiDiscrete(3,3,3,3,3,3,3,3,3,3,3,3)} \tag{4.2}$$

This second experiment is important as the final goal of the feasibility experiments is to check whether the lower level actions are feasible to use. The reason for this is that the higher level actions limit the decisions the PIRATE is able to make. The lower level actions translate to direct commands of the target positions and velocities of the joints and wheels, which allows the RL to figure out its own solutions. There exists an important trade-off between higher and lower actions. Compared to the higher level actions, the lower level actions have a more complex action space which increases training time, however the agent should have the availability to figure out better solutions. As mentioned earlier, the rotate section is left out. This is because defining lower level actions for each joint and wheel already increases the complexity substantially. Furthermore it would be an unfair comparison with the higher level actions, as there is no higher level action defined which is capable of rotating either.

Table 4.4: Showing the hyperparameters ranges used in the grid search Experiments.

Experiment 1	Experiment 2			Experiment 3	
Learning rate	Clip	Lambda	Entropy-coefficient	Train-batch-size	Mini-batch-size
[1e-4,1e-5,1e-6]	[0.1,0.2]	[0.99,0.95]	[0.005,0.01]	[500,1000,2000]	[300,100]

4.3 Hyperparameters Experiment

As mentioned in the analysis the population based training algorithm is not suitable to find proper hyperparameters. Due to this grid searches are performed, where the larger experiments are implemented with hyperband scheduling to speed up the experiment progress.

There are numerous RL hyperparameters. Due to time, not all hyperparameters are tuned. Experiments are done with the batch size, learning rate, clip, entropy-coefficient and the lambda. These hyperparameters are chosen as the intuition is that these hyperparameters have a larger influence on the experiments than others. The learning rate, and clip parameters are chosen to be tuned as these directly determine how much the weights of the value and policy function are updated. The entropy-coefficient could help to let the PIRATE explore, such that a better solution will be found. Furthermore, a large entropy-coefficient could form an unstable policy. This is simply because if certain actions are not allowed to dominate eventually, a solution may not be found. More elaborate information about the entrop-coefficient can be found in the original paper (WILLIAMS and PENG, 1991). The amount of exploration is also determined by the batch size. A small batch size will provide a larger variance in observations and thus introduce some noise. This noise could encourage exploration. A larger batch size will contain more values and thus more noise is cancelled out. However, some noise is needed to find a good solution. The increased noise by small batches could generalize the solution (Masters and Luschi, 2018). The lambda defines how much is bootstrapped. A lower lambda represents has a higher bias, but less variance. Where a higher lambda introduces less bias but a higher variance. Not all hyperparameters are tested simultaneously as this was time consuming on the current hardware (i7-4720HQ, 16GB, GeForce 940M). The disadvantage of this is that combinations of certain hyperparameters are not tested. This is justified as the goal of this thesis is not to figure out the optimal hyperparameters.

The first experiment searches for a proper learning rate. A grid search is performed with the following values: $1e{-}4, 1e{-}5, 1e{-}6$. After the learning rate a second experiment is performed with the entropy-coefficient, clip and lambda. The entropy-coefficient is chosen to be either 0.01 or 0.005, the lambda 0.99 or 0.95 and the clip parameter 0.1 or 0.2. At last an experiment with the train batch and mini batch is performed. For the train batch size, a size of 500, 1000 and 2000 are considered and for the mini batch size, a size of 100 and 300. To filter out fortunate experiments all experiments are sampled twice. The values are chosen based on the experience of some preliminary experiments. A summary of the experiments can be seen in Table 4.4

Common hyperparameters which are not tuned, are the configuration options for the neural network. These are for example the number of layers, the activation function and the number of nodes. These are not tuned because several neural networks are used in the upcoming experiments. Performing hyperparameter searches on each of these networks would be too time consuming. Some small experiments are performed such that there is some intuition about the number of nodes and layers. This is explained in Section 4.4.1.

4.4 Reinforcement Learning Experiments

After the feasibility training, more extensive experiments are carried out. The experiments are performed in a static and dynamic environment. In each of these environments, four experiments with different observation spaces for the PIRATE are performed. These experiments are

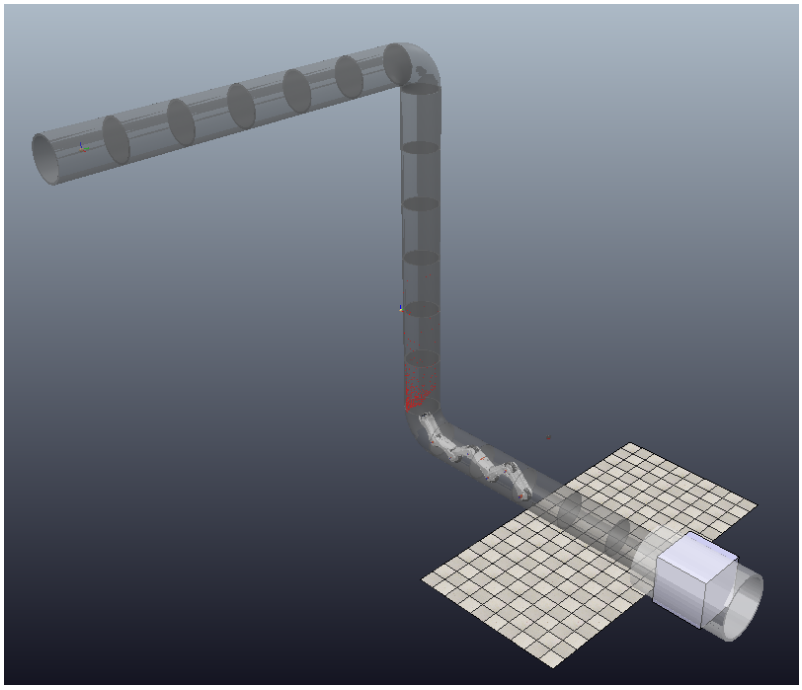


Figure 4.9: The V-REP static environment.

referred to as the Single Reinforcement Learning (SRL) experiments. In the next paragraph the different observation spaces for the different experiments are explained. After this, the static and dynamic environments are elaborated further.

The different observation changes in the observation space are applied in stages. In Chapter 3, three groups of observations were discussed, minimal observations, medium observations and vision observations. The four experiments are based on these groups. The minimal observations are defined as the observations which seem critical to figure out a RL solution. The minimal observations will be used for the first experiment. In the second experiment, the medium observations are used to see if the added observations could make a difference in performance. In the third experiment the vision observations are used. The camera should enable the detection of upcoming bends. The fourth experiment also uses the vision observations, but now an LSTM layer is implemented in the neural network. The LSTM should introduce memory, such that previous observations are incorporated for the agent.

The static environment is based on the environment of the feasibility experiments. The difference is that after going up, a 90° bend to the left is added. This is shown in Fig. 4.9. This forces the PIRATE to use its rotatable section as well. As the first experiments consider a static environment, each episode contains an identical environment. This should simplify the experiments, such that results are more easily realizable. Because it is a static environment the PIRATE might also learn how to make bends without camera observations. The PIRATE could learn to anticipate these bends entirely based on position observations, since the absolute position of these bends are always at the same place.

After the static experiments, the same experiments are repeated but in a dynamic environment. In this environment the pipe system changes each episode. Because the experiments are now dynamic, some of the created pipe systems could be easier or more difficult compared to the static pipe system. However, generally the dynamic experiments are more difficult as the PIRATE should generalize its solution. In this case, the experiments which include camera observations should be able to outperform the other experiments. This experiment should de-

Table 4.5: Hyperparameters used in experiments.

Hyperparameters:	learning rate	lambda	batch_size	minibatch_size	num_sgd_iter	clip_param	entropy_coeff
Value:	1e-5	0.99	1000	300	30	0.2	0.005

termine whether or not the camera observations are important for driving through corners. Of all observations, the camera observations might be the most interesting part.

Applying the observation changes in sequences, allows detecting of performance changes due to these changes. However, the changes can be applied in several ways. The above observations groups are chosen, to see the effect of these different observations. This allows one to check if the additions improve the model. The different observations groups expand on each other, such that more observations are introduced in each subsequent group. However these new observations could increase the importance of other observations as well. Thus, certain combinations of observations could improve the model. For example, joint orientation observations could become much more important when the camera observations are implemented as well. Although it would be interesting, to extend the observations from an LSTM or camera point of view, such that these changes would be visible as well, doing all these experiments would take too much time. The outcome of these results might give some insights such that new experiments can be determined. These new experiments could be part of future work. A flow diagram is shown in Fig. 4.10, which shows how the OpenAI Gym environment is configured for the different experiments.

To detect the differences between the observation space, the reward and action space should remain constant. The reward function used in these experiments is the main-reward. This reward function is explained in Chapter 3. The action space used is shown in Table 3.1, and is also explained in Chapter 3. As different observations spaces are defined for each experiment, the corresponding neural networks also differ. Figuring out the perfect network for each design is a time consuming process. The design of the neural networks is explained in Section 4.4.1.

The hyperparameters used are obtained from the results from the hyperparameters experiments, and are shown in Table 4.5. Other hyperparameters are configured by default or can be found in the included code. Although it is probably better to tune each experiment with there own corresponding hyperparameters, there is no time or computational hardware available to consider these design optimizations.

Something that is not mentioned yet, is when the environment is reset. It will of course reset when the PIRATE reaches its destination, but it should also reset when it gets stuck in the pipe system. It is quite difficult to detect when the PIRATE is stuck. It could for example still be learning and therefore not moving forward. Furthermore, it should have some time to actually try to figure out how it can free itself. Therefore, the following is implemented. The main-reward gives a positive reward when it is going forward and a negative reward when it goes back. In the episode all negative received rewards a summed up together. If the negative rewards exceeds a certain threshold the environment is reset. One might think that if the PIRATE is stuck no distance is traveled, and thus no negative rewards are received. Essentially this is true, but in reality the PIRATE is always moving a bit. Thus, still small negative and positive rewards are received. Eventually these small rewards accumulate and the environment is reset. Obviously, the threshold should not be to high to ensure the environment is reset in a reasonable amount of time.

4.4.1 Neural Network Designs

The first two experiments which include minimal observations and medium observations use a fully connected neural network. The first layer is a combined layer for the value and policy function. After the first layer, the neural network splits off to create a separate path for

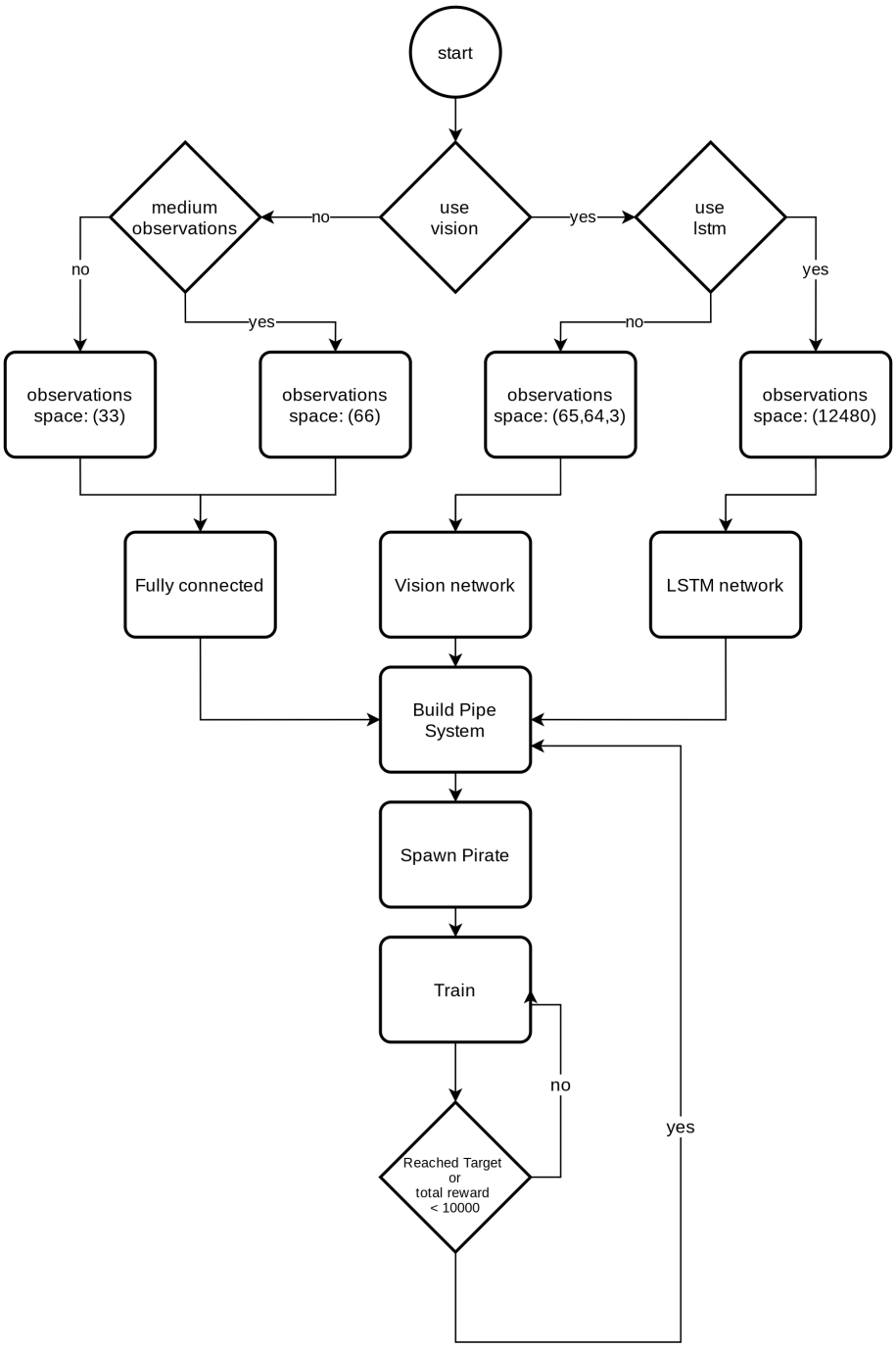


Figure 4.10: Environment Flow Diagram.

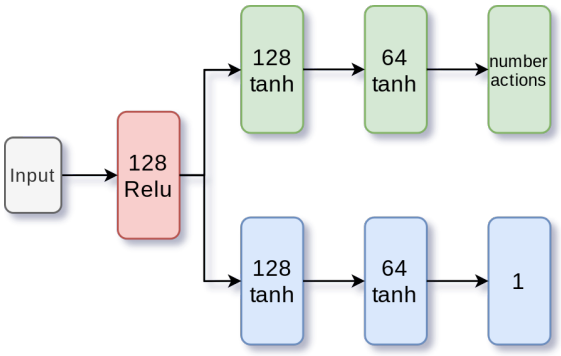
the value and policy function. A schematic of the fully connected neural network is shown in Fig. 4.11(a). The policy and value function both share the first layer to stimulate generalization on the first layer. After the first layer they can both represent there own functions, to potentially increase performance. The amount of layers and neurons in a neural network are difficult design considerations. A larger neural network is able to represent highly non-linear and highly-varying functions. However, many large layers could increase training complexity (Larochelle et al., 2009) and are more susceptible for overfitting on the training data. To see potential differences, experiments with the minimal observations group are performed where the number of layers is either two or three and the number of neurons in each layer is either 64 or 128. The results of these experiments are shown in Fig. B.1. The conclusion is that the results do not differ substantially for two and three layers. However, using a certain amount of neurons seems to stabilize the training a bit. A two layer network with 128 nodes and a three layer with 64 nodes seem to have more stable results. However substantially more experiments should be performed, as the results are not consistent. The first neural network only has 30 observations, for this a layer with 128 neurons is large. The upcoming experiments have more observations and for this reason probably more complex functions are required. This makes 128 neurons in each layer more reasonable. Based on the problem, it is also suspected that the overall function will be composed out of several smaller functions. As deeper networks are able to represent complex functions, a three layer neural network will be used. However, as already mentioned, a disadvantage of a large neural network is the ability to overfit a neural network. This would mean that the PIRATE is not able to generalize to other environments. To solve this, the PIRATE will learn in a dynamic environment. This encourages the algorithm to find a generalized solution. The static environment is already susceptible for overfitting as the environment does not change. As can be seen in Fig. 4.11(a) the activation function of the first layer is a relu function. The activation of the second and third layer is a tanh function. The reason the activation function of third layer is a relu, is because as more layers are added, the gradient of the tanh functions could become rather small. As the gradient with those of the previous layers are multiplied with each other for backpropation, the gradient could become really small. This results in slow learning for the the first layer. Changing this to a relu function, should increase the learning of the first layer, such that these layers also contribute in the network.

In the third experiment, the vision observations are used. In this observation space image data is added to the observations. The image data has a resolution of 64x64. A 64x64 image gives a sharp image without introducing a really big performance drop. A higher resolution decreases simulation and training performance. The image is first split off from the other observations. To extract and fed into two convolution layers. Two convolutional layers are assumed to be abundant as the amount of features within the pipe is small. After the convolution layers a dense layer is added such that features are reduced to 20 tensors. These 20 tensors are then combined with the same data input as the medium observations. The exact same network is applied after this concatenation. A schematic representation of this is shown in Fig. 4.11(b)

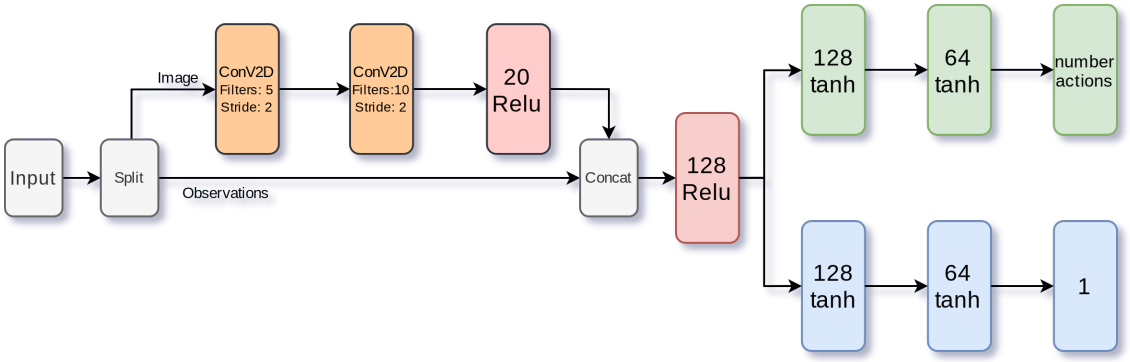
The fourth experiment contains the same observations as the third experiment, however now time batches with an LSTM are implemented. This requires changes in the neural network. A schematic representation is shown in Fig. 4.11(c). The Long Short-Term Memory (LSTM) is placed after the concatenation of the preprocessed image data. Placing the LSTM at the front of the fully connected layers means the processed images and original observations are fed into the LSTM. This architecture is based on (Hausknecht and Stone, 2017).

4.4.2 Sharp Corner

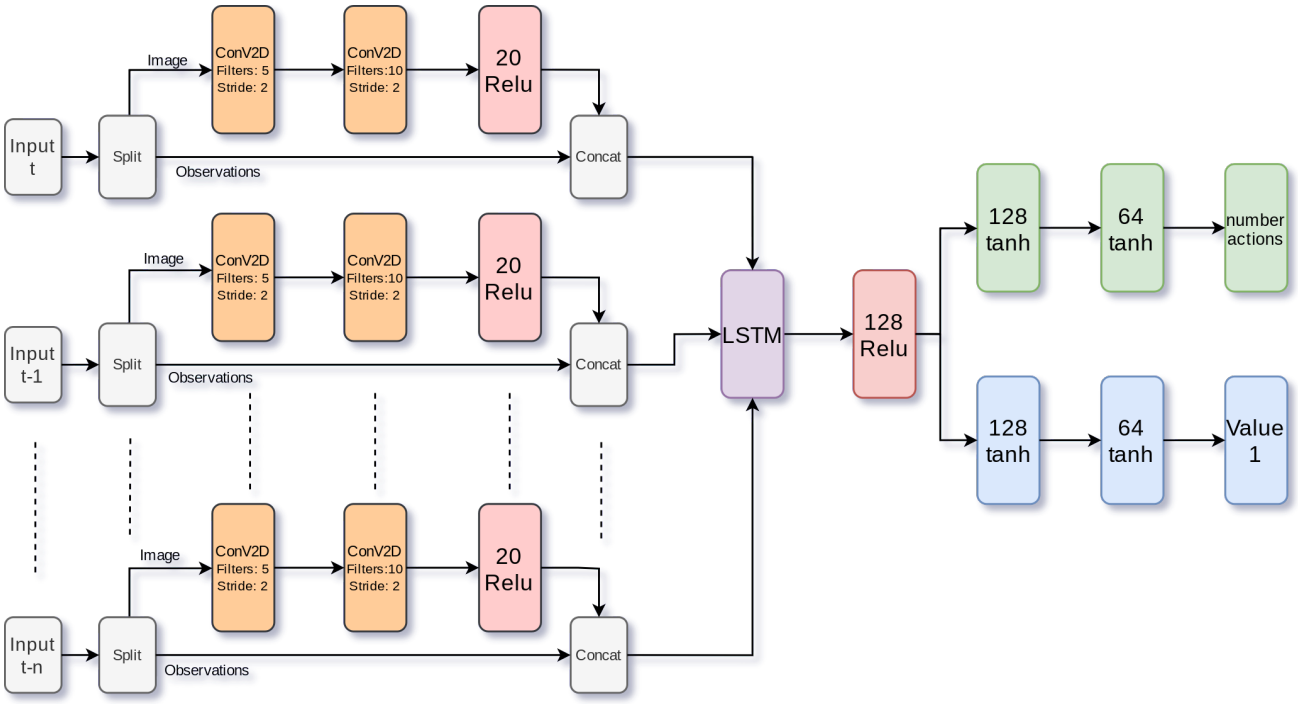
In the previous SRL experiments, the performance of PIRATE is evaluated in an environment like the one in Fig. 4.9. To extend the evaluation, experiments are performed in a larger dynamic environment consisting of five tighter bends. This should give an indication how the PIRATE



(a) Fully Connected Network.



(b) Vision Network



(c) LSTM Network.

Figure 4.11: Neural Network Designs.

performs in a more complicated environment. These tighter bends are shown in Fig. 4.12(b) and an example of the dynamic environment is shown in Fig. 4.12(c). The vision observations group is used in these experiments, with and without the LSTM implementation. The maximum time steps per episode are limited to 5000. If this is exceeded the environment will reset.

4.5 Hierarchical Reinforcement Learning Experiments

In this section the experiments and implementation concerning the Hierarchical Reinforcement Learning (HRL) are explained. The implementation of the HRL is explained in Section 4.5.1. Before the HRL experiment is performed, the sub-policies should be trained separately from the master policy. In Section 4.5.2, it is explained how the sub-policy experiments are performed. Section 4.5.3 then continues with the HRL experiment, in which the master policy controls the sub-policies.

4.5.1 Implementation

The Hierarchical Reinforcement Learning (HRL) is realized with the multi-agent capability of RLlib. RLlib provides a so-called multi agent environment (MultiAgentEnv[3]) which is able to handle multiple policies and agents. The concepts of the library RLlib that are important for this thesis are introduced in this section, however it is assumed that the reader is familiar with RLlib. Giving a detailed explanation of RLlib is not the aim of this thesis.

An important concept in the RLlib library are the policies[4], which are mentioned in Chapter 2. It is important that these policies of RLlib are not confused with the sub-policies defined in this thesis. The RLlib policies mainly exists out of the policy model. This shown in Fig. 2.7. In this thesis the policy model is a neural network, constructed with Tensorflow. In an OpenAI Gym (gym.Env[5]) environment there is only one agent and one policy. In a multi agent environment of RLlib there may be multiple policies each controlling one or more agents. This is shown in Fig. 4.13. Each agent has a separate Id which is linked to a certain policy. In this thesis there are four policies each controlling one agent. This is illustrated in Fig. 4.14.

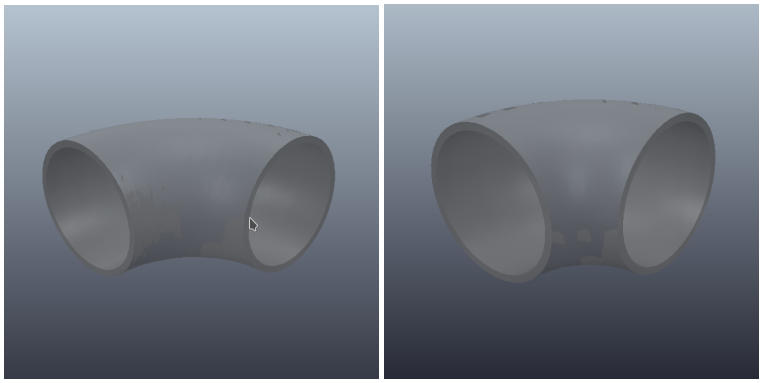
The common step function of an OpenAI Gym environment is implemented in the multi agent environment as well. The difference compared to the OpenAI Gym environment is that in the multi agent environment the argument of the step function is a dictionary. This dictionary contains as its key the agent Id, and as its value the actions. Just as the OpenAI Gym environment, the step function in the multi agent environment should return the following values: observations, reward and done. However, these values should now be put in a dictionary with the corresponding agent Id. This way the multi agent environment makes it possible to have multiple agents. In the configuration of the RL algorithm these agent Ids are linked to a certain policy.

To individually train the sub-policies they need their own OpenAI Gym environment. Thus, as these OpenAI Gym environments are already created for individual training, it is convenient to re-use parts of this code in the HRL implementation. In the implemented multi agent environment, the step functions of the OpenAI Gym environments, created for the sub-policies, are called if the corresponding policy is executed. The corresponding rewards and observations are then returned to the multi agent environment which links them to the corresponding policy. This means that in the HRL implementation, the OpenAI Gym environments do not function as an actual OpenAI Gym environment. However, they can be still used individually for training purposes. A schematic representation of this relation is shown in Fig. 4.14. The multi agent environment contains four agents, three of those agents are used to control the

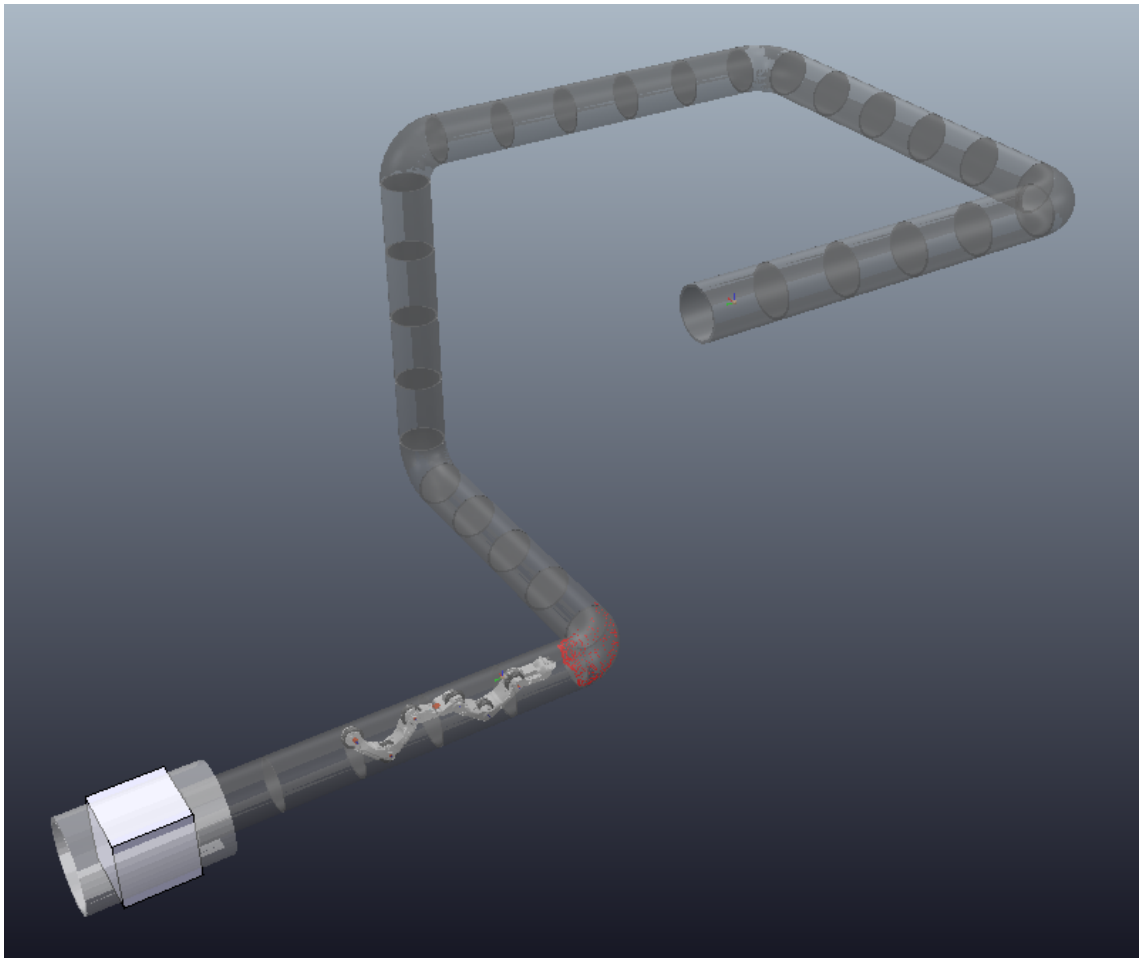
[3] https://docs.ray.io/en/master/rllib-package-ref.html?highlight=MultiAgentEnv#ray.rllib.env.MultiAgentEnv

[4] https://docs.ray.io/en/master/rllib-concepts.html#policies

[5] https://gym.openai.com/docs/



(a) First bend pipe section. (b) Sharper bend pipe section.



(c) The dynamic V-REP environment, used in the HRL and sharp SRL experiments.

Figure 4.12: V-Rep pipe system models.

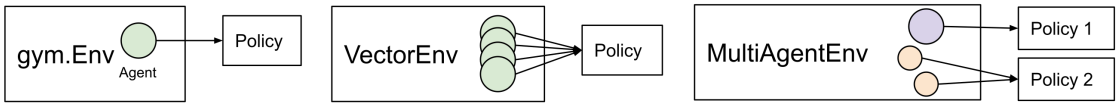


Figure 4.13: RLlib environments, agents and policies (Anyscale, 2020).

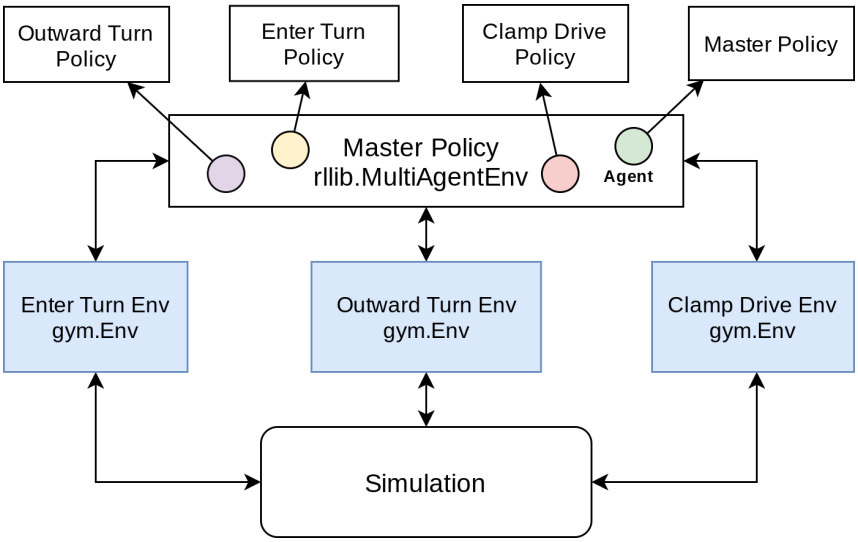


Figure 4.14: RLlib environment implementation.

sub-policies. One should note that the multi agent environment also receives its own actions and should return its own observations. Although technically there four agents in the implementation, the agent which is linked to the master policy decides which agent may perform its actions in the simulation.

The execution order of the HRL is shown in the sequence diagram in Fig. 4.15. In the diagram no specific sub-policies are mentioned for clarity. In the actual implementation there are three sub-policies. As mentioned, when the multi agent environment returns observations with a certain Id by its step function, actions from the corresponding policy are returned. In this HRL implementation only observations of one of the four environments are returned at a time. First observations of the master policy are returned, in the next step function a sub-policy is returned as its action. From that moment on observations corresponding to this sub-policy are returned, where actions of this sub-policy are returned. This will continue for 30 time steps. Then observations are again passed to the master policy which will choose a new sub-policy. This sequence will then happen repeatedly, until training is finished.

As the sub-policies are standard OpenAI Gym environments, they can be executed standalone for training. However, some custom implementations should be implemented such that these policies can be executed both ways. The most important change is that when the sub-policies are executed together, they operate with the same V-REP simulation environment. In a HRL implementation, this simulation environment is created by the master policy which then shares this simulation with the other sub-policies. If the sub-policies are operated solely as a standalone OpenAI Gym environment, they need to create this simulation environment themselves.

4.5.2 Sub-Policy Training Experiments

Training a HRL approach is a bit different then a normal RL environment. This is due to the fact that multiple policies exist. The master policy can only be trained properly if the sub-policies are already trained. This is really important as the master policy uses the sub-policies as its actions. If these actions are not converged to their optimal solution, the master policy might learn a different solution. If for example the enter-turn policy does not work properly, the master policy might choose to use the outward-turn policy to enter the turn. Just because the outward-turn policy is for example better at moving forward. The sub-policies experiments

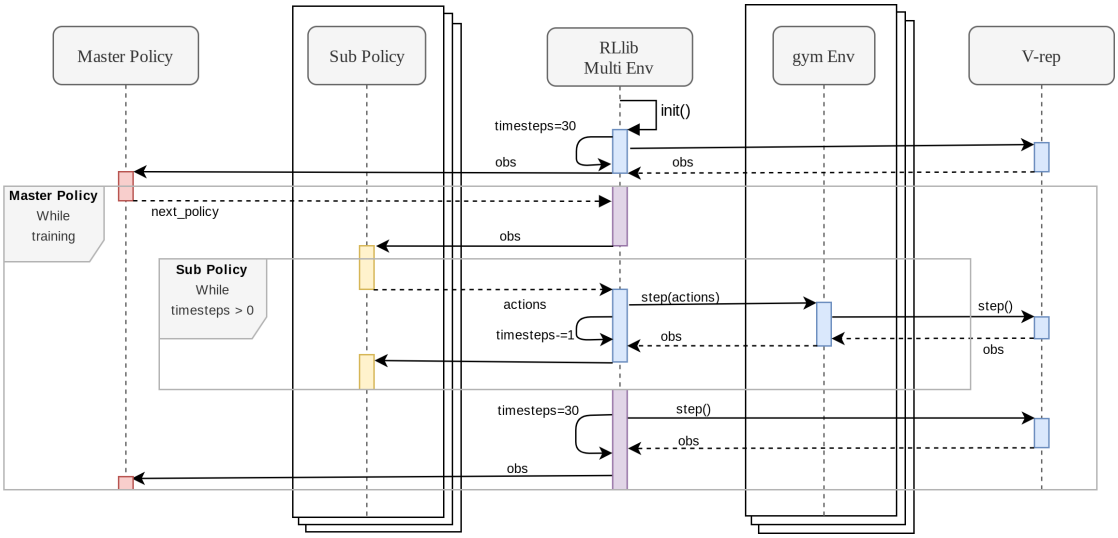


Figure 4.15: Time sequence diagram.

consist mostly out of how to properly train the sub-policies. Two methods are considered for training the sub-policies.

The first consideration is training all the sub-policies together within the environment. The master policy will randomly choose which sub-policy is executed and trained. The problem with this is that the for example the enter-turn policy can be chosen in a straight pipe section. At that moment it learns to drive through a straight pipe. Although it is not intended to do this, it will try to learn this. The same holds for the other policies, in other situations. The promise is that after training, the sub-policies will become relatively better at their specific task. The master policy can then discover which sub-policy it should use.

For the second consideration each sub-policy is trained individually. To implement this, each policy should be trained in their own domain. These domains represent the area for which the sub-policy is intended. The clamp-drive policy will thus only experience straight pipe sections. The enter-turn policy will only experience the entering of the turn. And the outward-turn policy will only experience leaving a turn. As theses domains are not static, the domain environments should be dynamic as well. Each domain is randomly placed in the world frame such that the PIRATE will not be able to depend on absolute positions. This is important when the sub-policies are used by the master policy in a larger pipe system. This is a more complicated setup then the first consideration, but will also probably be more effective. Each sub-policy is treated below to explain how the specific environments are implemented.

The clamp-drive policy is probably the most simple sub-policy as it should only learn to drive through pipes. These pipes should be randomized such that the PIRATE learns to drive vertically down and up. But also horizontal in several directions. The several directions are needed such that the PIRATE learns a generalized solution for the direction it is driving. It should learn that only the relative difference between the positions of the PIRATE matter. Other sub-policies could change the rotate section. Due to this, the clamp-drive policy also has to learn with a random initialized rotating section.

The enter-turn policy is always placed in front of a turn and ends when the policy reaches it target which is placed after the turn. The enter-turn policy only has to enter the turn with its front section. Because the sub-policy will enter several different bends, from different starting positions, it is important that the sub-policy will generalize as much as possible. Due to this, the turn direction and spawn place are randomized for training. The PIRATE needs to learn it cannot solely rely on position and orientation data, but should combine this with camera

observations. As mentioned in the Chapter 3, two different experiments are performed for the enter-turn policy. The first experiment will be performed with vision observations and the second with an LSTM layer.

The outward-turn policy starts at the ending position of the enter-turn policy. It then should be able to drive out of the pipe. The same reasoning as with the enter-turn policy, also holds to generalize the outward-turn policy.

As already mentioned in Chapter 3 the master policy can choose from three sub-policies. Each of these sub-policies contain there own neural network. The neural networks used in the SRL are reused. As explained in the Chapter 3, all sub-policies need the camera observations. Due to a lack of time, the same hyperparameters are considered as the previous experiments. These are shown in Table 4.5.

4.5.3 Master Policy Implementation

In this implementation the master policy chooses a sub-policy. This sub-policy is executed for 30 time-steps. A fixed time-step termination is used as it is quite difficult to implement an actual terminating condition based on the progress of a sub-policy. There was simply no time to research these better termination conditions. A relatively short time-step is chosen to increase the amount of observations the master-policy receives. This also ensures that the master policy can change to a new sub-policy when required. The sub-policy has not a shorter time step then 30 because otherwise the sub-policy has to little time to change its posture and actually perform some actions. If the sub-policy does not get a chance within one action step to show its effectiveness, the master policy could reject certain sub-policies more easily as they seem useless.

The master policy receives the vision observations as it also needs to know its current state, such that it can decide the next sub-policy. The master policy also receives two more observations. These observations are the previous sub-policy and the previous reward. The previous reward could help in predicting the next sub-policy. If for example less reward is obtained, the master policy might consider a different policy. This should prevent that the neural network will overfit to one policy. An example of using a previous reward can be found in Espeholt et al. (2018). The reward structure of the main policy only consist of main-reward. This should be sufficient as this reward represents the main goal of driving forward through the pipe.

4.5.4 Hierarchical Reinforcement Learning Experiments

In the HRL experiment the trained sub-policies are used as the actions for the master policy. In these experiments the sub-policies are not trained anymore. The performance of the HRL with the vision observations is compared to its performance with the neural network with the LSTM.

This means that the master policy uses the neural networks shown in Figs. 4.11(b) and 4.11(c). The timebatch length of the LSTM observations should not be too long, as the master policy will then receive irrelevant observations. However, the timebatch length should also not be too short, since then the master policy is unable to discover a certain action sequence. An extensive grid search to determine an optimal value for the timebatch length is out of scope of this thesis. A timebatch with a length of 15 is chosen based on the number of times the sub-policies are required to drive towards a turn, enter this turn, leave this turn and drive again. A timebatch of this length should be long enough for the policy to discover pattern to correctly take a turn.

In the sub-policies the hyperparameters of the SRL are used as the experiments are similar. In the master policy collecting batch sizes similar to the SRL would take a long time. Therefore, the batch size of the master policy is 33 and the mini batch size is 10. This roughly means that the size compared to the SRL is decreased by factor of 30. As the length of sub-policies take

30 time-steps, this roughly compensates the learning rate. Learning with smaller batch sizes is also good for the generalization of the master policy (Masters and Luschi, 2018). This is also mentioned in Section 4.3.

The HRL experiments are performed on the same dynamic environment used last for the SRL, shown in Fig. 4.12(c). A more complicated pipe system makes performance differences between the HRL and SRL clearer.

4.6 Conclusion

In this chapter the design and implementation is explained. This section describes the most important implementations and the designed experiments. All experiments are performed by the combination of the simulator V-REP and the RL framework Ray. One of the reasons the Ray framework is used is because of the implemented support for multi-agent environments. The performed experiments can be categorized into 4 groups, the feasibility experiments, the hyperparameters experiments, the SRL experiments and the HRL experiments. The purpose of feasibility experiments to figure out whether the constructed pipe system and PIRATE are feasible. This is done by performing a manual controlled experiment as well as with RL. After the feasibility experiments, several experiments are performed to find good hyperparameters to be used in later experiments.

In the SRL experiments the different observations groups are tested and compared in a static and dynamic pipe environment. The minimal observations group and medium observations group use a fully connected neural network. The vision observations group contains depth images such that a different neural network is required. This neural network network contains a separate section to extract the features of this depth images by convolution layers. A fourth experiment is performed by introducing a LSTM layer using the vision observations group.

To elaborate on these experiments a larger dynamic environment with tighter turns is used. On this environment only experiments with the vision observations group with and without the LSTM are used to research the performance. To improve the shortcomings of the SRL experiments, a HRL experiment is performed. The sub-policies are trained in their related domains. For the enter-turn policy this is for example entering a turn. To realize separate training, each sub-policy is made with its own OpenAI Gym environment. When the sub-policies are trained they are used in the HRL implementation. The HRL experiment is compared to the SRL implementation containing the vision observations in the larger dynamic environment.

5 Results and Discussion

In this chapter the results of all experiments are presented and discussed. In Section 5.1 the results of the feasibility study are shown and discussed. In Section 5.2 the results of the hyperparameter experiments are shown and discussed. In Section 5.3 the results of the Single Reinforcement Learning (SRL) experiments are shown. This section also contains a separate discussion of the dynamic and static experiments. In Section 5.4 the Hierarchical Reinforcement Learning (HRL) experiments are presented and discussed. Finally there is a general discussion in Section 5.5 which contains not earlier mentioned results and discussions, from all performed experiments.

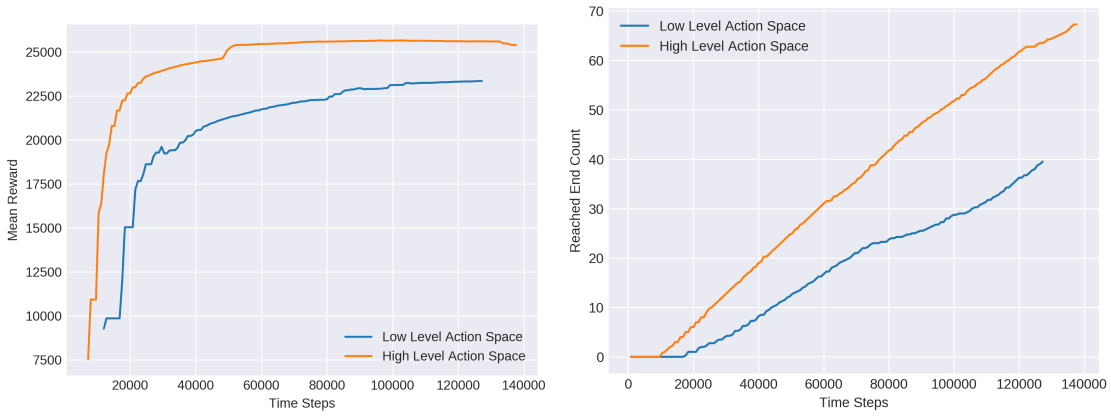
5.1 Feasibility Results

The results of the feasibility experiment are shown in Fig. 5.1. Both the experiment with the lower level actions and the experiment with the higher level actions find a solution. The experiment with the high level action space learns faster and also performs slightly better in comparison to the experiment with the low level action space. This is probably because the lower level action space is much smaller, compared to the higher level action space. Furthermore, the agent with the low level action space has to figure out how to combine certain actions to accomplish driving and clamping. These action relations are already hard-coded in the high level action space. As mentioned in Chapter 3, learning with lower level actions takes more time. The agent that uses the high level actions only has to figure out which high level action it has to apply in which part of the pipe system. Moreover, the reason it also retrieves slightly more reward is probably because the high level actions are optimized for the current pipe system. The agent that uses the lower level actions is able to drive forward, but struggles to find a good method to go through the bend. Given enough time to learn, the agent with the low level actions space might eventually perform just as well as the high level actions agent. However, the low level actions agent might not find the same higher level actions that are defined for the high level actions agent. If the defined higher level action space is close to the optimal solution, the experiment with the lower action space can do only worse. Although, the agent with the lower level actions performs slightly worse and learns a bit slower, the difference with the PIRATE with the higher level actions is not as striking as expected. With the lower level action space the PIRATE still learns reasonably fast, considering the more complicated action space. The results of the feasibility experiment show great promise as both configurations find a solution quite fast. Thus, this gives confidence that the actions can be partially replaced for a continuous action space.

5.2 Hyperparameter Results

The results of the hyperparameter experiments are shown in Fig. 5.2. These graphs are based on actual results shown in Figs. B.2 and B.3. In Fig. 5.2 the average result of each hyperparameter is shown.

First the results of the learning rate experiment are discussed. These are shown in Fig. 5.2(a). The result for each learning rate in Fig. 5.2(a) is based on two trials, which can be found in Fig. B.2. The results of the learning rate are very clear. The reward of the experiment with a learning rate of 1e−4 does not improve at all. This indicates that the learning rate is too high. In one of the experiments with a learning rate of 1e−6, it starts to figure out a solution around 400000 time steps. This exact trial can be found in Fig. B.2. However, this occurs much later than with a learning rate of 1e−5, which starts learning directly. Something that has to be noted, is the drop-off of the orange trial in Fig. B.2. This is probably not related to the specific learning rate, but to the behavior of the PIRATE. This is discussed later in this chapter.



(a) Mean Reward

(b) Reached End Count

Figure 5.1: Results of the feasibility experiments.

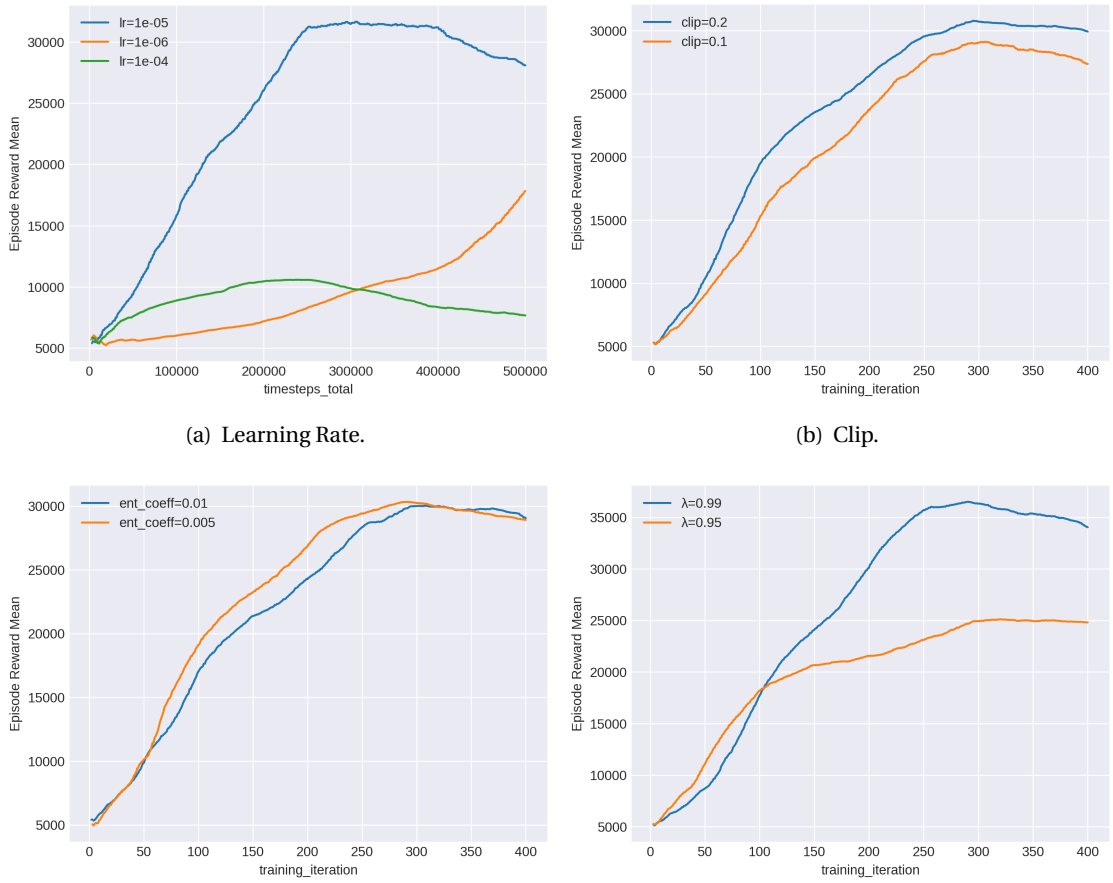
The next experiment is a grid search of the lambda, clip and entropy coefficient. As this required 16 separate trials, plotting the results in one figure is not comprehensible. Thus, the results in Fig. 5.2 are separated for each hyperparameter shown in Fig. B.3. The results in Fig. 5.2 are of combined trials such that they can be easily compared. The trials which were stopped by the hyperband were not taken into consideration. The difference between the results of the different clip and entropy parameters are minimal and thus no clear optimal value can be indicated. The best performing value is selected for the next experiments. However, a clear result is present for different values of lambda. A higher lambda value of 0.99 significantly improves the performance. Updating the value function with less bias and more variance seems to have positive effect on the learning rate.

The last experiment tries different values for the mini-batch-size and the training batch size. The results are shown in Fig. 5.3. The results are shown together as taking the average of experiments with different lengths due to the different batch sizes is not possible. A small training batch size of 500 is performing poor. Furthermore, a size of 2000 seems to large. A size of 1000 seems to perform better. Selecting a proper mini batch size based on the results is a bit more difficult as it is not entirely clear what performs best. However, a training batch size of 1000 and a mini batch size of 300 seems to perform the most consistently as both samples have high rewards.

From the results it can be concluded that the hyperparameters have a significant effect on the performance of the PIRATE. Especially, the learning rate parameter has a large influence whether a solution will be found or not. Dedicating some time to find promising hyperparameters is definitely worth it. However, for an extensive search to optimize the hyperparameters, more powerful hardware is required. The Tune implementation could be easily scaled up to more powerful searches.

5.3 Single Reinforcement Learning

In this section the results of the static and dynamic experiments of the Single Reinforcement Learning (SRL) are shown and discussed. In Section 5.3.1 the performance of the different experiments in the static and dynamic environment are discussed based on created mean reward graphs. In Section 5.3.3 several things not directly related to the results of the previous section are discussed. Finally in Section 5.3.4 there is a conclusion.



(a) Learning Rate.

(b) Clip.

(c) Entropy coefficient.

(d) Lambda.

Figure 5.2: Grid search of learning rate, lambda, clip and entropy coefficient.

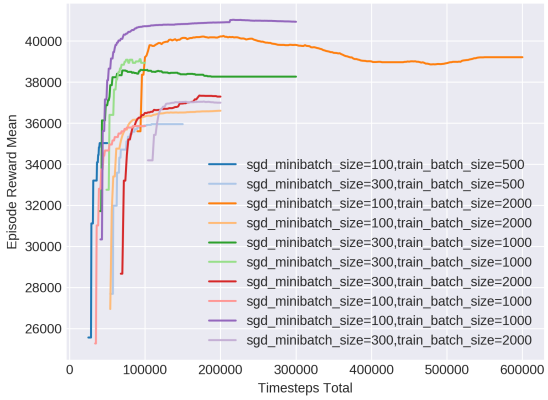


Figure 5.3: Batch size grid search experiment.

5.3.1 Results Single Reinforcement Learning Experiments

For both the static and dynamic experiments five figures are shown, see Figs. 5.5 and 5.6. The first four figures show the results of the four experiments which respectively use the minimal, medium, vision and LSTM observations groups. The last figure combines these experiments in one figure such that the results can be easily compared.

Static Experiments

In this section the static experiments are discussed. Each experiment will be discussed separately. After this they will be compared with help of the last figure.

In the first experiment, with the minimal observation group, the pink trail 41e77c63 achieves more reward but then falls off. This is shown in Fig. 5.5(a). There is actually a simple explanation for this. In the YouTube video, which is provided in Fig. 5.5(a), can be seen that the PIRATE clamps with its back section, where its front section is straight. With this posture the PIRATE is able to rotate its front section, such that it can learn how to orient itself for a turn. Being able to rotate depends a lot on how the PIRATE learns to clamp. If it is clamping as in the bottom image of Fig. 5.4, it will be much more difficult to rotate. With minimal observations it will be harder to detect if it is in this position. If it is clamping as in the top image of Fig. 5.4 rotating will be easier. This problem is experienced several times, out of clarity it will be from now on referred to as the clamp rotate problem. After more training time, the PIRATE in the pink trail 41e77c63 starts to clamp with its front section as well. This makes rotating much more difficult. This explains the fall off in reward. It is not entirely clear why the PIRATE starts to clamp with its front section. It might be because it needs more friction to drive upwards and thus it learns to clamp with both sections. Moreover, it could be due to the fact that driving with two sections clamped becomes easier in all straight pipe sections. The lowest performing green trail 41e77c62, simply does not learn to properly rotate for the upcoming corner, even when its posture seemed to allow this. If the PIRATE does not learn to rotate, it will try to push itself through the corner.

In the second experiment, with the medium observations group, similar conclusions can be drawn as in the first experiment. However, when the PIRATE learns how to rotate it is more stable. For example, the pink trial 01b637b2 remains at a mean reward close to 39000. This trial is probably more stable as its more aware of its exact posture in the pipe, than with the minimal observations group. The different performance of the trials can be explained by the fact that in the lower performing trials, the PIRATE learns to clamp in a sub-optimal way. The main-reward function does not specify how the PIRATE should figure out a solution. This should give the PIRATE the freedom to find a good method. The problem with this, is that there are several ways the PIRATE could clamp to achieve its goal. However, not all clamping configurations are equally well. In order to learn to drive through a corner the PIRATE first should reach this corner, which requires learning how to clamp. If it clamps in such a way that it is not able to rotate properly, it will be hard to take the turn. This is referred as the clamp rotate problem. Although the agent will still explore, chances are small that it will discover a new solution to clamp. As the episode restarts, it receives rewards for clamping in the current approach such that its action distribution becomes even smaller. And trying to explore to clamp in a better way would result in a significant drop in reward. Hence, this does not happen.

The third experiment, which incorporates camera observations, clearly outperforms the other experiments. The reason for this is because more information that can be used to enter a turn properly is available, since positional data of the PIRATE does not give direct feedback about how the environment actually looks like. The brown trail 5317df31 already performs optimal at 200000 time steps. The other trials catch up slightly later. In comparison to the previous experiments, the PIRATE is to able to orient itself such that it can enter the second turn smoothly. It tries to do this without bumping in the corner. Looking at Fig. 5.5(e) it can be seen that includ-

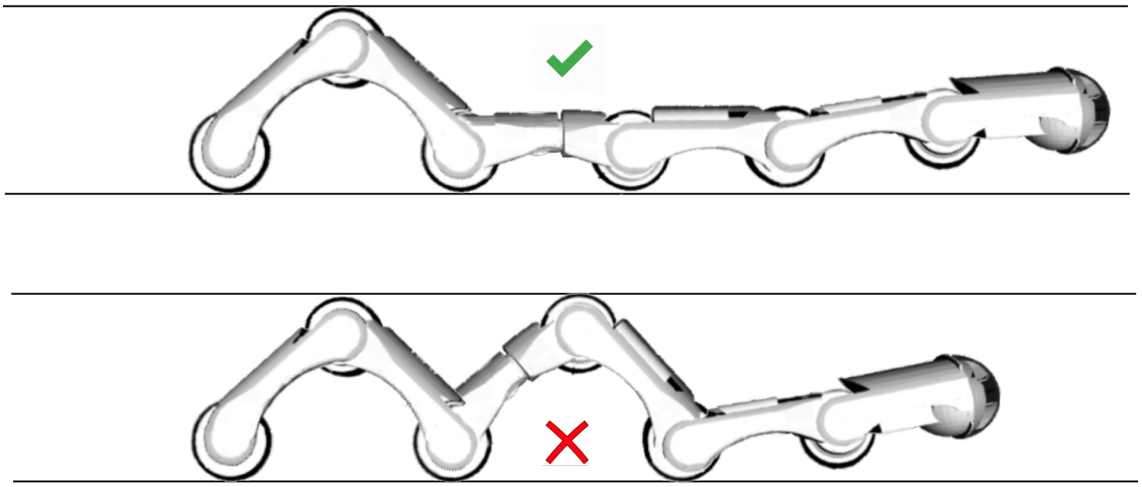


Figure 5.4: Problem that arises if the PIRATE learns to clamp in a wrong way. If it clamps with its front section rotating is difficult.

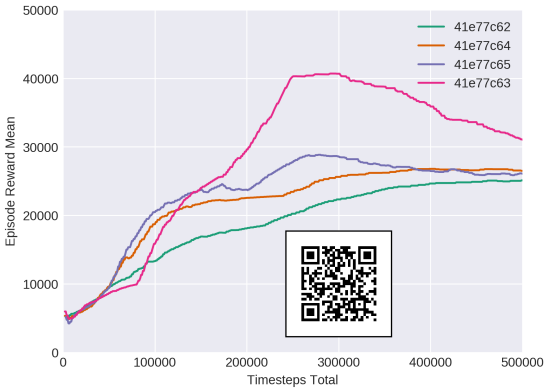
ing the camera stabilizes the mean reward. The camera observations thus allow the PIRATE to discover the relationship between the environment and its own orientation. Of course, without the camera observations it can still rotate, but lacks insight into finding a helpful relation with the environment.

In the last experiment the neural network with the LSTM layer is used. From Fig. 5.5(e) it can be concluded that the PIRATE learns the fastest with the neural network which has an LSTM layer. The LSTM helps for learning how to clamp. This makes sense, since a clamping operation is sequence of actions. Although less training steps are required, a training step with an LSTM layer takes considerably longer, due to the high amount of trainable parameters. A Youtube video of trail 5a5cd511 at time step 225000 is provided in Fig. 5.5(d). In this trial the reward is around 40000, as mentioned earlier with such a reward, the PIRATE performs well. As with the experiments that included vision, this trial does not bump the bend such that it can go through. In fact, when required it drives a bit backwards, such that it can properly orientate itself. However, after more training the mean reward drops, which can be explained by the clamp rotate problem. The green trail addc4c6c at time step 500000 performs considerably less. This is because the PIRATE is quite cautious to perform the turn. This can be due to the fact that the value function is updated when the PIRATE gets stuck, such that there is a lower probability it will perform actions such that it gets stuck. The neural network with the LSTM layer is much larger compared to the other neural networks used in the experiments. Therefore, it might need more than the 500 training iterations. Especially as the green trial addc4c6c has probably not converged yet. Furthermore, reducing the gamma could help in making the PIRATE less cautious.

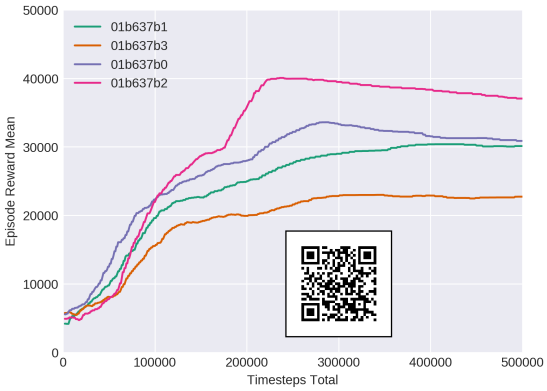
Dynamic Experiments

The results of the dynamic experiments are shown and are compared to their corresponding static results.

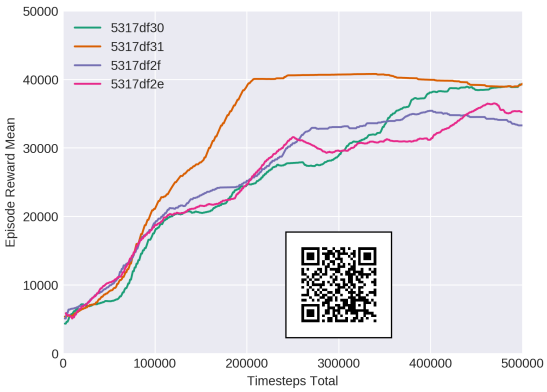
Unsurprisingly, the first experiment with minimal observations does not perform as well as its counterpart in the static environment. The PIRATE is bumping into the turn to figure out in which direction the turn is going. With this approach it actually manages to go through corners. The pink trial 3b3747a7 in time step 200000 clamps mostly with its back section such that it can



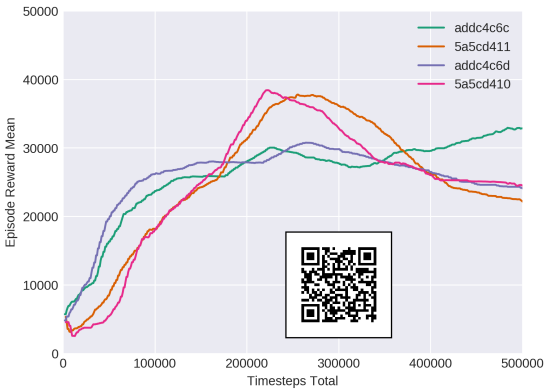
(a) minimal observations experiment, the qr code shows trial id: 41e77c62 at Time Step 500000 and trial id: 41e77c63 at time step 300000.



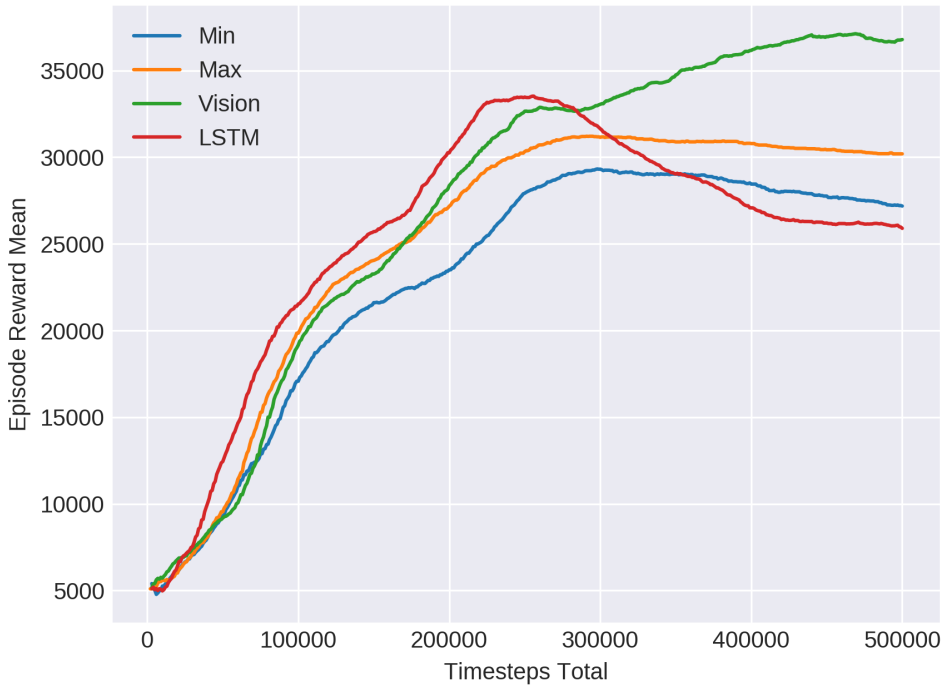
(b) medium observations experiment, the qr code shows trial id: 01b637b3 at Time Step: 500000 and 01b637b2 at Time Step: 300000.



(c) vision observations experiment, the qr code shows trial id: 5317d41 at Time Step 300000.



(d) Experiment with LSTM layer, the qr code shows trial id: 5a5cd411 at Time Step 225000 and trial id addc4cc6c at time step 500000.



(e) Compare

Figure 5.5: SRL, Static environment results.

easily rotate. This can be seen in the YouTube video provided in Fig. 5.5(a). After 200000 time steps, the mean reward starts to drop. The reason for this is that the agent starts to clamp more with is front section. This prevents it from easily rotating the front section when it bumps a bend. This results in it getting stuck more often such that the reward drops. This also referred to as the clamp rotate problem.

The second experiment performs better then the first experiment at time step 200000, but drops off afterwards. Similarly to the first experiment at time step 200000 it clamps with its backside, and navigates through corners with the front side. Due to the extra observations the PIRATE is more aware of its current posture. Again, the decrease in mean reward after a while is because the PIRATE starts to clamp with its front side. This can also be seen in the YouTube video in Fig. 5.5(b). The decrease in reward is more noticeable compared to the static environment, because the environment is dynamic, which makes proper rotating more important.

In the third experiment vision observations are introduced. The mean reward is not much higher than the best experiments of the first and second experiment, but the decrease in mean reward after a while is less present. With the vision observations, the PIRATE orients itself before the corner. In this experiment there is larger difference between the trials. This is because some trails are able to rotate better. The brown trail 58a2aefd clamps more with its front section, which prevents proper rotating and discovering the usefulness of the camera observations. The green trial 300fa58f does not clamp with its front section and can thus rotate more easily. Entering a turn most of time goes quite smoothly, however leaving the turn not. It sometimes gets stuck and tries to push itself through the corner, without trying to properly orient the back section of the PIRATE.

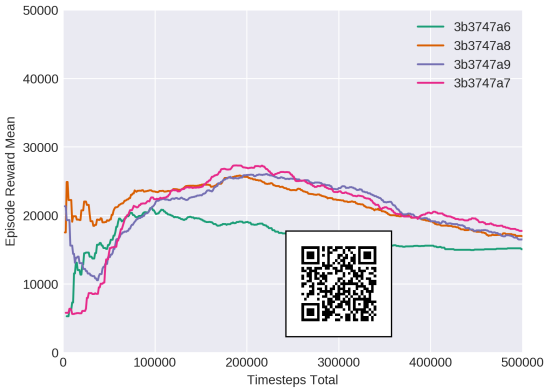
In the fourth experiment the neural network with the LSTM is introduced. The PIRATE with an LSTM layer also seems to learn faster in the dynamic environment. The mean reward is similar as for the vision observations, but is more stable around time step 200000. A YouTube video is made of the brown trial d1f19570 at time step 500000. In this trail the same behavior is shown as in the vision experiment. The PIRATE is able to orient itself such that it can enter the turn, but can become stuck when it tries to leave the turn. Also, the PIRATE behaves a bit more stable and less wobbly in the environment, then without a LSTM layer. The reason for this is probably because it can sense previous actions better. Comparing this to the reward of the static environment it is actually quite similar. In the dynamic experiment the LSTM is less cautious and goes through corners with more confidence, even if this means it will bump the pipe a little bit.

5.3.2 Sharp Corner Environment

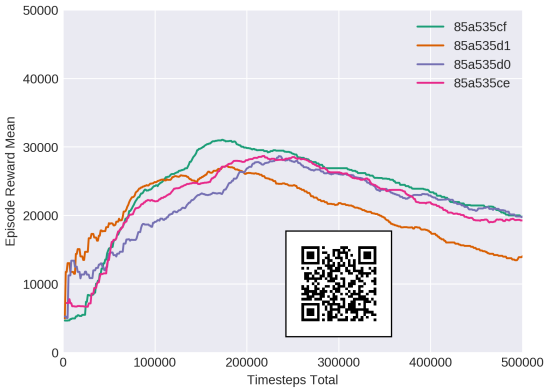
In this section the results of the sharp corner experiment are shown and discussed. In Fig. 5.7 the results are shown. There is a big difference between the two best performing trails and the two lowest performing trials, although overall the performance is poor. The poor performance can once again be explained by the clamp rotate problem. In all trails the PIRATE is not able to drive through the environment. The sharper corners compared to the previous experiments make driving through the corners much more complicated. In Fig. 5.7(a) there is a qr code with a YouTube video. In this video can be seen that the PIRATE is not capable of driving through the corner. Furthermore, the average episode lengths are shown in the Fig. B.4. The LSTM seems to perform a bit better by average episode length, but after inspecting actual trials, the PIRATE is not able to make proper turns.

5.3.3 Discussion Single Reinforcement Learning Experiments

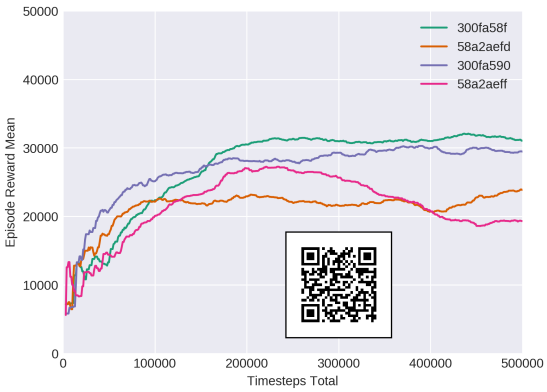
In this section final thoughts about the SRL experiments are discussed. Each paragraph contains a discussion about a separate topic.



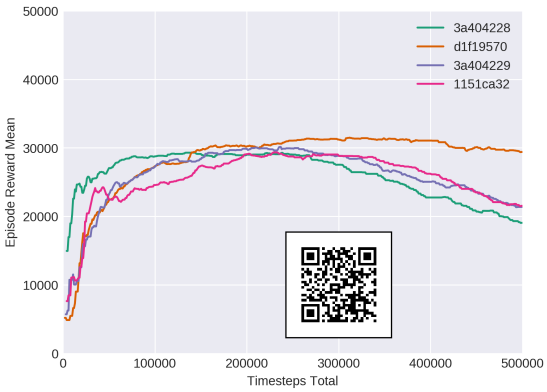
(a) minimal observations experiment, the qr code shows trial id: 3b3747a7 at Time Step 200000 and trial id: 3b3747a7 at Time Step: 500000.



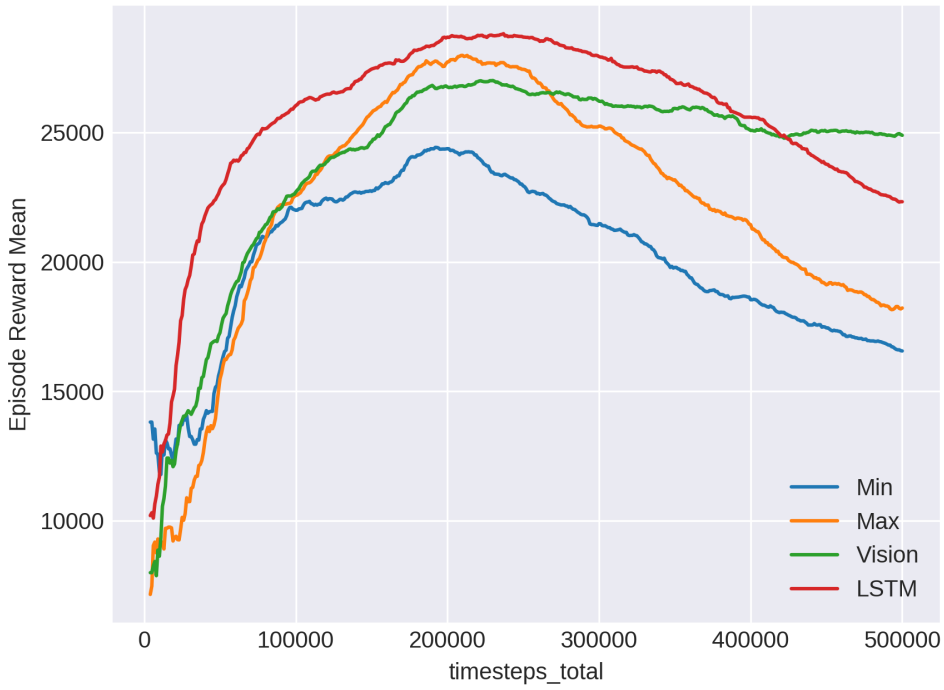
(b) medium observations experiment, the qr code shows trial id: 85a535cf at Time Step 200000 and 500000.



(c) vision observations experiment, the qr code shows trial id: 300fa58f at Time Step 500000.

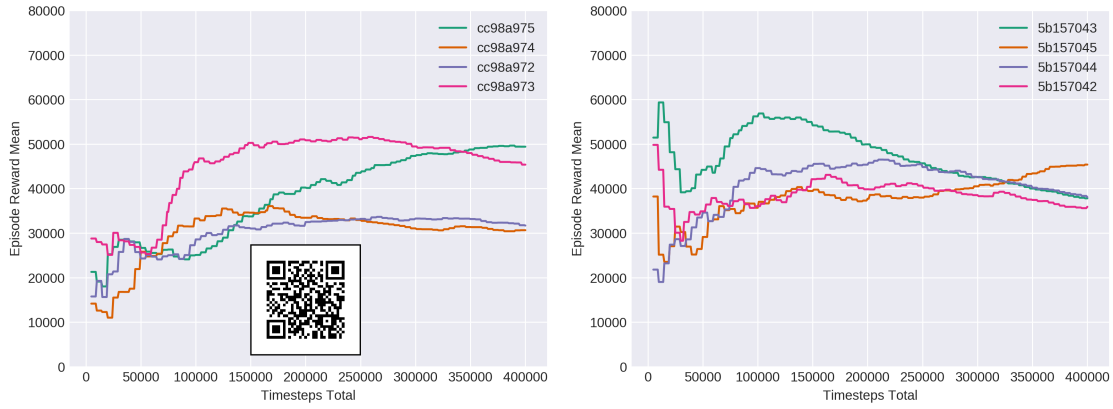


(d) Experiment with LSTM layer, the qr code shows trial id: d1f19570 at Time Step 500000.



(e) Compare

Figure 5.6: SRL, dynamic environment results.



(a) SRL results.

(b) SRL results.

Figure 5.7: SRL sharp corner results.

Comparing the different experiments based on the observations changes alone is not really justified, because the experiments also include different observations sizes with different neural networks. One of the experiments might have a better network or better hyperparameters for its specific case. For example, the experiment with the vision observations might be better tuned then the other experiments. Furthermore, some results from performing the same experiment multiple times differ greatly. Performing multiple samples of the same experiment of course helps to show this problem, but some experiments still deviate a lot to make good conclusions.

Another interesting thing to note, is that including the LSTM layer in the neural network introduced new behaviors. The PIRATE never drives backwards to free itself, when it got stuck in a bend. However, with the LSTM layer the agent sporadically tried to drive backwards such that it could free itself from the corner. This is observed in the dynamic and static environments. Furthermore, the PIRATE was able to perform more complex behaviors, such as multiple configurations to enter a turn. However, these behaviors did not always introduce better performance. Something that should be considered is that the neural network with an LSTM layer simply needs more training or better tuned hyperparameters.

Although the results of this experiment conclude that the camera is working. Faster hardware is necessary such that many parallel hyperparameter experiments can be performed. The amount of experiments which are eventually performed might be to low to draw conclusions. Additionally, the different experiments can be tuned to their best behavior.

In the current design the agent resets when it receives too much negative reward, for driving backward. When it gets stuck it receives only small negative rewards, it can then take quite some time to reset. Initially it was thought that the PIRATE should get chance to free itself. As this would mean that a more robust solution can be found. However, learning how it can free itself might not be an efficient approach. This is precious learning time which could be spend, for example, to learn not to get stuck in the first place. Thus, implementing an algorithm which resets the PIRATE when it is stuck might improve the learning rate.

It is also interesting that the PIRATE often does not use its full clamping capability, but is only using its backside. As already mentioned, this is because it is then able to enter a turn more easily. Furthermore, the amount of friction and torque available could make clamping with two sections less important, as clamping with one section could be sufficient. The actual torque and friction on the actual PIRATE could be lower. Also the friction of the pipe itself, and the fluids that the pipe will normally carry, play a huge roll in clamping. Nevertheless, this result is

interesting as the real PIRATE is also unable to clamp both sections when it needs to perform certain maneuvers.

It should also be mentioned that the dynamic environment is not only more difficult because the turns are random, but also because it has to perform turns in which it is driving downwards, which makes the PIRATE more susceptible to get stuck due to gravity.

5.3.4 Conclusion

The maximum reward for each episode is approximately 40000. However, most of the experiments in both the dynamic and static environment do not achieve this reward, or become unstable after more training. This is because the PIRATE gets stuck for a certain amount of time, or takes more time to orient itself for the turn. Due to this the PIRATE does not make it till the end of the environment, and receives less reward. For the static environment, the experiment with the vision observations performs the best. Where in the dynamic environment the experiments with the included LSTM layer performed the best. Both experiments include camera observations and thus this clearly improves the performance, however it is not a complete solution. This can be clearly seen in the experiment where a larger environment and tighter bends are used. This clearly showed that the current solution is not able to handle tighter turns and it is getting stuck most of the time.

In the dynamic and static experiments the PIRATE still gets stuck. The main problem is that learning to drive forward can interfere with learning to enter a turn. The better the PIRATE learns to drive through a straight pipe by clamping its sections, the harder it becomes to enter the upcoming turn. The camera observations can mitigate this problem by making the agent aware of the upcoming corner. It is then able to figure out that rotating its front section is important. Entering turns in the dynamic environment becomes more difficult compared to the static experiment, since in a dynamic environment it can be hard to figure out that rotating has a relation with the camera observations.

The experiments without the camera simply do not know that a corner is approaching. The only way to experience the corner is by its current posture. However this means the PIRATE should already be in the corner. This makes finding a solution even more problematic. It is surprising that the experiments without the camera do figure out a solution by bumping in the corner to deal with a dynamic environment. However, bumping a corner to determine in which direction it goes is not sophisticated solution.

Leaving the turn properly, is not really smooth in any of the experiments. When the PIRATE is halfway through the turn, the PIRATE could get stuck when the back section is not oriented properly. The PIRATE does not discover that it can turn its back section if it clamps the front section of the PIRATE, such that it can leave the turn more easily.

5.4 Hierarchical Reinforcement Learning

In this section the results of the HRL experiments are shown and discussed. In Section 5.4.1 the sub-policy results are shown. In Section 5.4.2 the master policy results are shown. At last there is a discussion in Section 5.4.3.

5.4.1 Sub-Policy Training Results

The first training proposal was training all the sub-policies within the same environment. In Fig. 5.8 the results are shown. The mean rewards obtained by the sub-policies are quite similar. This is probably because the reward per time step are quite similar and the trials all learn to drive at a similar moment. Furthermore, when the PIRATE gets stuck, the trials all receive less reward, which explains the dip in reward at similar moments. After around 200000 time steps the mean reward seems to stabilize and the sub-policies do not increase in reward anymore.

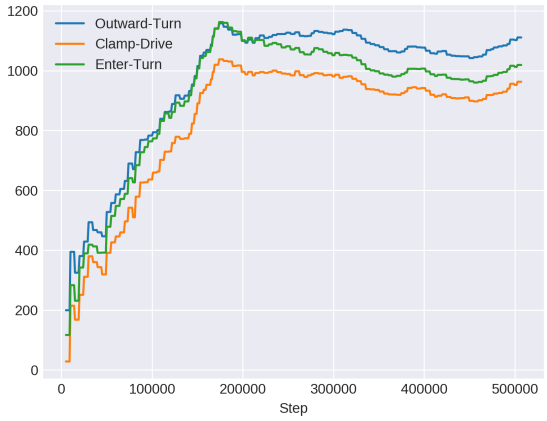


Figure 5.8: Shows the mean reward while training all sub-policies together with a random master policy.

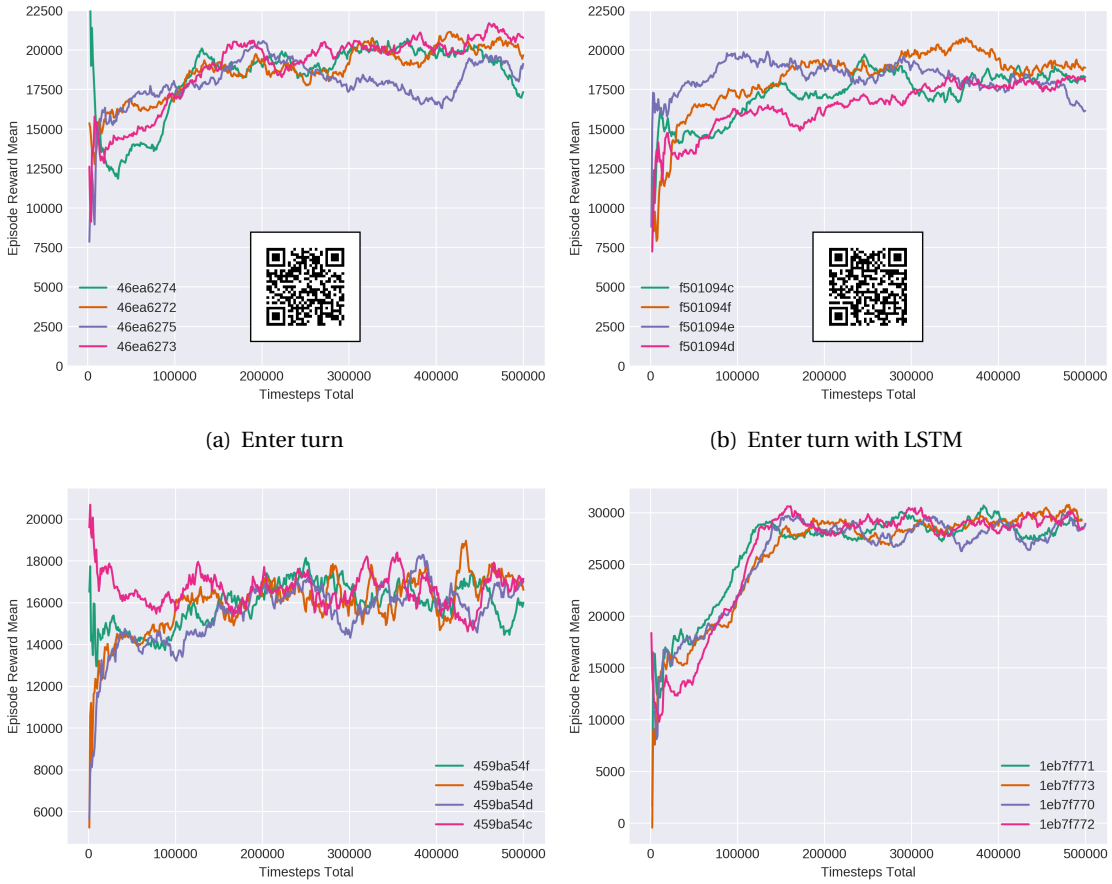
The actual performance of the sub-policies is poor. Especially the enter-turn policy does not properly learn how to enter the turns. The sub-policies basically generally learn how to drive, but do not specialize to their intended purpose.

The second training proposal, in which the sub-policies are trained separately, gives better results. Something to note is that the graphs and mean rewards cannot be compared to the simultaneous training as they are performed in a complete different setting. As can be seen in the graphs in Fig. 5.9, most trials have quite similar performance. This can be explained by the addition of the intrinsic rewards and a less complex environments. Due to the intrinsic rewards most trials end up with a similar solution. The high rewards in the beginning of some trials are due to an episode in which the PIRATE falls down with a correct orientation of the bend. Furthermore, the average episode lengths are shown in Fig. B.5.

The enter-turn policy is trained with and without an LSTM layer. The experiment without the LSTM layer seems to reach its maximum reward around 200000 time steps, which can be seen in Fig. 5.9(a). The mean reward only rises minimally after this. Trail 46ea6275 even starts to drop after 200000 time steps. This can be explained by clamping the front section, such that entering a turn becomes more difficult. Because the front wheels are not actuated there should not be an advantage to clamp the front section. It might be that by clamping the front section a bit more grip can be achieved such that driving through the pipe becomes easier. The experiment with the LSTM has a more unstable reward and based on the trails it is not entirely clear at which point the maximum reward is achieved. Also the episode length is not clearly converging. Furthermore, the maximum reward is less then without the LSTM. Also based on the YouTube videos the experiment with the LSTM layer seems to perform worse.

The clamp-drive policy reaches its maximum reward around 150000 time steps. All trials seem to perform rather similar. By inspecting the results in V-REP, the PIRATE is always able to reach its target. The inconstancy of the mean reward can be explained by the fact that the sub-policy consists out of several sub-rewards. Especially, the depth-reward is responsible for these fluctuations.

The graph of the outward-turn policy is not really clear as the some trials already have a large reward at the beginning of training. This can be explained by the simpler dynamic environments where the PIRATE almost does not have to learn something to achieve a high reward. An example of such a simple environment is when the PIRATE has to drive out of a corner, initially coming from a downwards vertical section. In this situation gravity helps to move the PIRATE in the correct direction. However, the average episode length clearly converged around 200000 time steps.



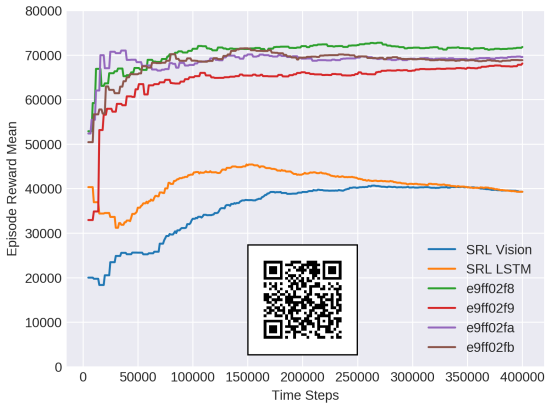
(a) Enter turn

(b) Enter turn with LSTM

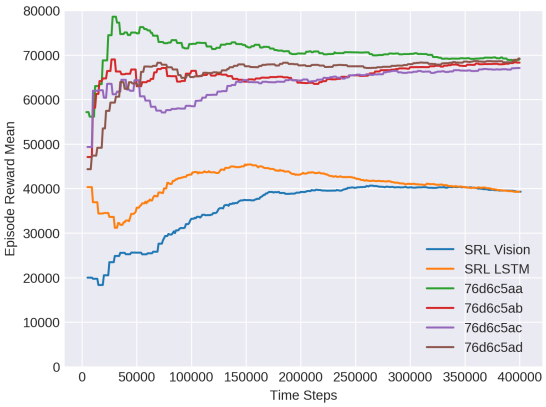
(c) Outward Turn

(d) Clamp Drive

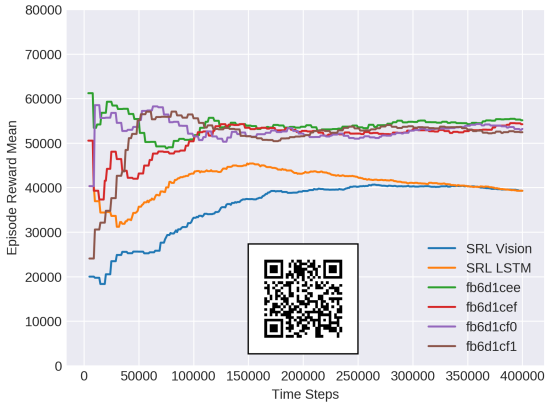
Figure 5.9: Separate training: Sub-Policy Results.



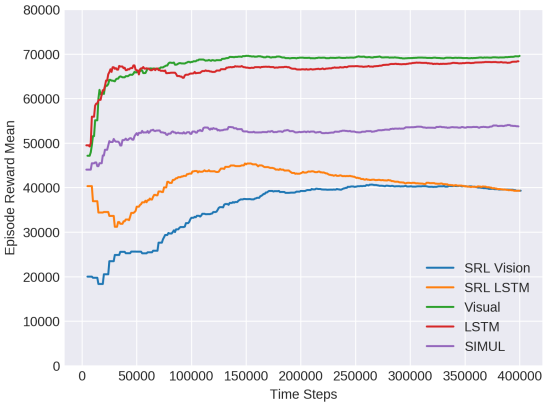
(a) Compare the SRL to the HRL trials without an LSTM layer for the master policy.



(b) Compare the SRL to the HRL trials with an LSTM layer for the master policy.



(c) Compare the SRL to the HRL trials where the sub-policies are trained simultaneously.



(d) Compare the SRL to the average HRL trail results with and without the LSTM layer for the master policy.

Figure 5.10: HRL Results.

5.4.2 Master-Policy Training Results

The results of the master policy are shown in Fig. 5.10. In each graph the average of the sharp SRL trials are shown. These are the same results presented in Section 5.3.2 and will be used for comparison as they are performed with the same reward function and in the same environment as the HRL. In Fig. 5.10(a) the average of these SRL trials is plotted alongside the trials of the HRL without an LSTM layer for the master policy. In Fig. 5.10(b) similar results can be seen of the HRL with an LSTM layer for the master policy. The results show that the mean reward of the HRL experiments quickly converge to approximately 70000. The average mean reward of the SRL is around 40000. However, inspecting the SRL results separately, shows that there are two trials which actually achieve a mean reward of 50000. This is still significantly less then the performance of the HRL. Also the performance of the master policy combined with sub-policies which are trained simultaneously are added, which is called SIMUL and shown in Fig. 5.10(c). The performance of this experiment is better than the SRL experiments, however much worse then the master policies which use sub-policies which are trained in a separate environment. The average performance of all experiments is shown in Fig. 5.10(d). Furthermore, the average episodes lengths are shown in Fig. B.4. Here can be seen that some trials actually still improve after reaching an average maximum reward.

5.4.3 Discussion Hierarchical Reinforcement Learning Experiments

In this section the HRL experiments are discussed. By evaluating and discussing results, suggestions for future work occur as well.

The HRL experiment achieves better results than the SRL experiments. The main difference is that the HRL solution is able to successfully and consistently move through corners. By randomly changing the absolute positions and domains of the simulation environment, the sub-policies seem to be generalized. Although, the HRL experiment performs better, the master policy still often makes wrong decisions and due to this the PIRATE gets stuck. This means it could take a while before the correct sub-policy is chosen. It could be that the master policy simply is not trained properly, and different hyperparameters should be used. Another explanation would be that the sub-policies simply are not specialized enough to see a clear difference for the master-policy. Something that should be noted is that although the reward function and distance is the same for the HRL and SRL, the amount of time steps a reward is obtained differs. The time steps of the HRL are longer which makes the calculated distances longer. With these longer distances the total distance has higher chance of being effectively shorter, as the SRL might wiggle a bit through the pipe such that the distance is a bit longer then the actual distance driven. Due to this, the actual performance between the SRL and HRL might be bit larger.

In Chapter 3 is mentioned that the master policy might benefit from an LSTM layer. As shown in the results, the performance of the master policy with and without the LSTM is similar. As mentioned in the previous paragraph, the master policy does not seem to be properly trained and this could be the reason for the LSTM as well. Another possible reason might be that the observations of the master policy differs to much to recognize a pattern for the LSTM layer. The observations of the master policy contain several scalars, were only only one of those scalars represents the current sub-policy. One might argue that the other observations should contribute to states which contribute to recognizing a current domain as well. However, as the master policy only receives observations of certain time-steps it could be difficult to distinguish these states. The LSTM layer is complicated layer and more elobarate research might be required to determine whether an LSTM layer can improve the performance in this situation.

Due to time and hardware limitations, no elaborate experiments are preformed to find the optimal values of the hyperparameters. As there are many hyperparameters it is very unlikely that the master policy is optimally tuned. As the hyperparameters could determine whether a solution could be found, more experiments should be performed.

Experimenting with a HRL setup requires more manual tweaking compared to SRL. What absolutely should be prevented is that a sub-policy is able to perform outside of its intended domain, since it would make it difficult for the master policy to converge. It is quite hard to prevent this as all the sub-policies are more less trained to drive through the pipe system. When a wrong sub-policy receives reward for an unintended domain, the chance this sub-policy is taken increases. The master policy then becomes less likely to discover the proper sub-policy. To make things worse, in some situations a sub-policy used in the wrong domain could actually perform better then the intended sub-policy. An example of such a situation would be if the PIRATE is aligned correctly with the bend of a pipe, and the clamp-drive policy is used. In this situation the clamp-drive policy will actually perform better then the enter-turn policy because it has more grip and more actuating wheels. This also holds for sub-policies which in the short term might receive some reward. What is meant by this is that if for example the clamp-drive policy is able to drive a little bit into the turn, the master policy still receives some reward. The correct enter-turn policy might actually drive a little bit backward such that it can actually orientate for the turn, however the feedback for the master policy would be a negative reward. The master policy should explore enough such that it will discover that for entering the the turn it should stick with the enter-turn policy.

After performing several experiments with the hierarchical structure, it can be concluded that using sub-policies which blend nicely together is crucial. For example, if the master policy switches from the clamp-drive policy to the enter-turn policy the bottom section should clamp in the same way. If one of the sub-policies clamps in different way, the PIRATE looses its grip when it is changing its joints. If this happens in a vertical section of the pipe system the PIRATE will fall down.

In the analysis was mentioned that sparse rewards can make learning more difficult, as well as multiple reward can conflict with each other. This was indeed the case as the combination of depth-reward and main-reward rewards make learning more difficult. Naturally the reward is still a scalar, but finding the accompanying behavior to maximize both reward functions is more difficult.

A problem with the SRL is that the PIRATE becomes stuck in a corner. Although the hierarchical agent is less likely to get stuck, due to the dedicated turn policy, the PIRATE could become stuck in the hierarchical experiments as well. This occurs either by performing a wrong sub-policy or simply by failing at the task. Something that was noticed when performing the hierarchical experiments is that due to the switching of policies the PIRATE can actually free itself. The sub-policy that got the PIRATE stuck is often unable to free the PIRATE as it tries actions which made the PIRATE become stuck in the first place. Another sub-policy performs different actions, such that the current state of the PIRATE changes. Even if the changed sub-policy is not able to move forward, it could change the current posture such that the next sub-policy could free itself. Although, the exact behavior cannot really be specified, it is an advantage of HRL over SRL.

As mentioned in Chapter 3 it is important that the sub-policies observe the pipe system such that the master policy receives helpful depth images. If this is not the case, it will be harder to determine which sub-policy the master policy should use. Of all the sub-policies, the outward-turn policy struggles the most with this requirement. This is because by bending the camera section upwards, more torque can be created such that it can clamp in a better way. However, by doing this the camera cannot look through the pipe anymore.

In the current HRL experiment only three sub-policies are included. Better performance might be achieved by including more sub-policies. For example, to avoid that sub-policies clamp in a different way, the direction in which to clamp is defined. However, this prevents the possibility to clamp in the other direction. In some situations this might give a better solution. The amount of sub-policies could be extended to incorporate these reverse sub-policies, or sub-goals could be provided to the existing sub-policies such that the master policy can decide in what way the sub-policy should clamp. This can be considered for future work. Something that was also noted in Chapter 3 was the ability to include regular actions along the sub-policies. This is not considered as is out of scope of this thesis. However, after experiencing the performed experiments, including these actions in a proper way might be extremely difficult. Also it is likely that the master policy does not see the benefit of these actions and simply rejects them.

Defining the sub-policies for the different scenarios turned out to fit quite well. There are two domains which caused some trouble. The first domain occurs when the PIRATE leaves a turn with its backside hanging vertically down. The weight of the PIRATE puts a lot of stress on the clamp mechanism, which makes driving out of the turn quite a hard objective, especially as the wheels of the backside are disabled. The reality is that the outward-turn policy aligns the backside of the PIRATE, such that the clamp-drive policy is able to drive out the bend. Although this might be a solution, this was not intended. This also stimulates the master policy to learn to use a sub-policy in the wrong domain, which makes learning more complicated. Furthermore, this solution is a bit more unrealistic, especially as the corner would become more right-angled, and the clamp-drive policy is not able to drive out of the turn. The second domain is when the

enter-turn policy should enter the turn from a downwards vertical position. Since it is falling down, it can easily become stuck this way. Due to gravity it should actually drive a bit upwards to prevent going downward to fast. Because driving to the corner is rather easily, the PIRATE has less time to orientate itself correctly for the turn. The PIRATE is able to make small corrections such that it can free itself in this situation, but when the orientation is completely off it is unable to free itself most of the time.

In the HRL experiments, the same neural network architectures are used as in the dynamic and static experiments. As the sub-policies operate in their own domain and in principle should fulfill a less complex task, the neural networks could be smaller. This could increase the learning rate of the sub-policies.

5.5 Discussion

In this section there are some general discussions not mentioned in previous sections. Furthermore, also in this discussion some future work is considered.

All experiments are highly dependent on the model designed in V-REP. Especially the amount of friction, torque, mass and inertia determine the outcome of the RL solution. A more realistic model can definitely be made, however this is not the focus of this thesis. A more realistic model might actually make the RL implementation easier. For example by providing a way to include springs in the model and locking the current position of the joints, as this makes it possible to provide a better way to clamp the joints. This would also prevent that the wheels can affect the posture of the PIRATE, which is explained in Section 4.1.3. Furthermore, the HRL experiment could benefit of more realistic models. The sub-policies behave a bit unrealistic due to the torque controlled clamping. If the joints would behave sturdier, the clamp-drive policy might not be able to go through a corner. This would stimulate the master policy to use only the enter-turn policy and outward-turn policy in corners. Furthermore, a more realistic model could facilitate transfer learning to the real world.

In the static and dynamic experiments the main-reward is used. As mentioned in Chapter 3, this reward function gives a reward for moving forward, and it actually gives more reward for moving faster. Based on this reward function one might expect that the PIRATE would let itself fall in a vertical section of the pipe. Based on the SRL experiments this is actually rarely the case. A possible explanation for this might be that letting itself fall will not increase the amount of reward per episode, but the chance it gets stuck increases significantly. Skillfully driving downwards such that it has more chance to go through the corner and also receive rewards after the turn, seems like a better approach. Another potential explanation is that it is simply too hard to distinguish to the complete different approach that is necessary to obtain more reward when it has to potential to fall downwards.

Neural networks have a lot of configuration options. Most of these configuration options are not explored in the current thesis. Potential performance increases could by made by adding and tuning: regularization, dropout layers, activation functions, learning method, loss function and initializing kernel values. Examining the potential benefits of using these techniques is simply not achievable with the current hardware. Especially as most of the mentioned techniques have their own hyperparameters.

The step size of the simulation is 50ms. The RL algorithm is currently using this same time step. As discussed in the analysis multiple actions per second could actually increase learning difficulty. As the time steps are so small the actions barely have the time to settle. It will probably be a good idea to decrease the amount of actions taken by the agent, such that observations differ more and the consequences of certain actions become clearer. This can be done by running multiple simulation steps between choosing new actions. However, by changing the amount of actions per second, model parameters and hyperparameters might need new tuning.

The weights of a neural network are by default initialized with samples of a uniform distribution[1]. It is observed that the initial weights of the network could influence the final outcome of the RL solution. Initially sampled weights determine a certain action distribution, this action distribution could promote to clamp in certain direction. Due to this, there is higher chance the PIRATE actually learns how to clamp based on these initial values. This could influence that some trials learn faster and better then others.

[1] https://www.tensorflow.org/api_docs/python/tf/keras/initializers/GlorotUniform

6 Conclusion

In this chapter the conclusion of this thesis and additional recommendations for future research are outlined.

6.1 Conclusion

The aim of this thesis is to research the requirements to autonomously drive the PIRATE in V-REP. Before, the main research question is answered, the four sub-questions are discussed.

1. Which observation groups can be beneficial for the PIRATE?
In this thesis three observation groups are examined, namely the minimal observations group, the medium observations group and the vision observations group. These observation groups are defined in Chapter 3, however they are shortly described in this section again. The minimal observations consist out of joint positions, absolute wheel positions, joint orientations and previous actions. The medium observations group contains all observations which are in the minimal observations group and joint velocities, wheel velocities, more joint orientations and more absolute wheel positions. The vision observations group consists of the observations in the medium observations group and vision observations of a depth camera. These observations are tested in a dynamic and static pipe system environment in V-REP. The experiments with the medium observations group performed better compared to the experiments with the minimal observations group. Furthermore, the experiments with the vision observations group performed better compared to the experiments with the medium observations group. The vision observations performs better as the depth information is critical to enter turns properly.

However, it is not straightforward to determine exactly which observations in the vision observations group lead to the enhanced performance. The fact that a neural network is a black box is a well-known issue. This is because the back propagation algorithm could set certain weights to zero. Thus, to answer this question in more depth the weights of the neural network should be examined. However, examining a neural network is no easy task. Determining which individual observations are relevant for the PIRATE is out of scope for this thesis.

2. Which reward functions can increase the performance of the PIRATE?
Rewards based on the Euclidean distance between the PIRATE and an end target do not provide a feasible solution. This is because Euclidean distance between the PIRATE and an end target could increase while actually moving forward in the pipe system. Therefore the main-reward is designed, which is explained in Chapter 3. Briefly explained, the main-reward measures the difference of the distance between two time steps of PIRATE. Essentially this means that the reward is equal to the distance driven in the pipe system. From the experiments in this thesis, it follows that this main-reward performs well. Thus, reward functions based on this concept are likely to work reasonably as well. For example, the main-reward function could be altered such that the reward increases exponentially over the distance traveled. However, this could also have adverse effects. This alteration will make combining this reward function with other reward function more difficult. Furthermore, because the loss which is based on the value function is minimized, using an exponential reward function might actually be redundant. An advantage of using a distance based reward is that experiment performances can be easily compared.

In the HRL experiments several sub-reward functions are used for the sub-policies. The considered sub-reward functions are the depth-reward, the stretch-reward and the clamp-reward. The depth-reward should prevent the camera from pointing to the side of the pipe, which

would make the camera observations unusable. The stretch-reward is intended to stretch the front part of the PIRATE, such that entering a turn is easier. The clamp-reward is indented to stimulated the PIRATE to clamp a certain section. The depth-reward performed well as it ensured that the camera pointed straight through the pipe. Furthermore, the clamp-reward also worked as indented. The stretch-reward does not function properly as it could not prevent the PIRATE from clamping its front section.

3. To what extent can an LSTM improve the performance of the PIRATE?
Several experiments are performed with a neural network which included an LSTM layer. If the answer to this sub-question would be purely based on the mean reward from in the graphs in Figs. 5.5 and 5.6, the answer would be that it could improve the performance of the PIRATE. However, it could also make the experiments more complicated. The LSTM layer is a complex layer which might need a better tuned RL algorithm. Furthermore, the PIRATE seemed to perform more complex behavior when the neural network contained an LSTM layer. For example, the PIRATE used multiple postures. However, this behavior lead to decrease in performance, but a properly tuned policy which is able to handle complex actions could easily outperform most of the experiments.

4. To what extend can a hierarchical action structure help in improving the performance of the PIRATE?
In this thesis RL experiments are performed with a hierarchical structure. This structure contained one master policy and three sub-policies. These sub-policies are explained in Chapter 3, but for convenience are briefly outlined here. The enter-turn policy is intended to enter a turn. The outward-turn policy is used to leave a turn. The clamp-drive policy is used to drive through straight section of the pipe. The sub-policies performed well and introduce skills which are not easily discovered by single RL experiments. Actually, in the SRL experiments no combinations of the used sub-policies are observed. The sub-policies introduce skills such as entering and leaving the turn which are required for a proper RL solution. Thus, an hierarchical action structure does improve the performance of the PIRATE. However as mentioned in the discussion of Chapter 5, it does make the RL training considerably more complex.

> *Research question: What is required to fully autonomously drive the Pipe Inspection Robot for AuTonomous Exploration (PIRATE) within the simulation environment V-REP by Reinforcement Learning?*

The answer to the research question is naturally based on the performed experiments. Using the vision observations group enhances the performance of the PIRATE compared to the other observation groups. Also using observations from a simple depth sensor with a low resolution of 64x64 is adequate. More advanced sensors will probably only perform better. Furthermore, basing the reward function on the distance driven works well. However, if the PIRATE should drive to a certain position in the pipe system, this type of reward function is not sufficient. SLAM algorithms could provide a solution such that distance or target based reward functions through the pipe itself can be used. However, this is out of scope of this thesis. Furthermore, a HRL solution gives the ability to divide the problem into multiple sub-problems. As autonomously driving through the pipe system is a complex problem, dividing the problems with a HRL solution is essential. However, using sub-policies introduces new complications. Moreover, the LSTM implementation might be helpful, but makes the implementation of the RL more difficult. Better computational hardware might be a solution to overcome these difficulties. Also better computational hardware opens up the possibility to better tune the hyperparameters of the RL algorithms.

In this research the PIRATE was able to complete the pipe system robustly. However, more complex pipe systems can be designed and the current performance can be increased. The research showed that that a Hierarchical Reinforcement Learning solution can make several complex tasks manageable. This gives confidence that the PIRATE can also be be fully autonomous in more complicated pipe systems by using a hierarchical RL structure.

6.2 Additional Future Recommendations

Apart from future recommendations mentioned in the discussion, this last section contains additional suggestions for future work. Each paragraph in this section will contain a suggestion.

When a working model is realized, the agent could be optimized concerning other important aspects, such as energy consumption. Punishing the PIRATE for using too much torque or electricity could stimulate the PIRATE to figure out more energy efficient methods.

Something that would be interesting is dynamically changing the dimensions of the pipe. The expectation is that this should not make the problem much more difficult. Furthermore, it could be interesting to change the size of the PIRATE itself as well. The size of the PIRATE could for example represent a scalar such that the RL could change its behavior according to the size of the model.

The master policy receives scarce observations as the sub-policies take several time-steps to complete. The camera observations are important for the master policy to differentiate between different situations. To improve the camera images the camera could be actuated by a separate joint, which would then be controlled by an individual sub-policy. Furthermore, to increase the reliability of the master policy observations it might be possible to use several time steps of observations to determine the next the sub-policy. This will improve the probability that useful depth observations are obtained.

A Appendix 1

$$\begin{aligned}\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\ &= \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) \\ &= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})\end{aligned} \tag{A.1}$$

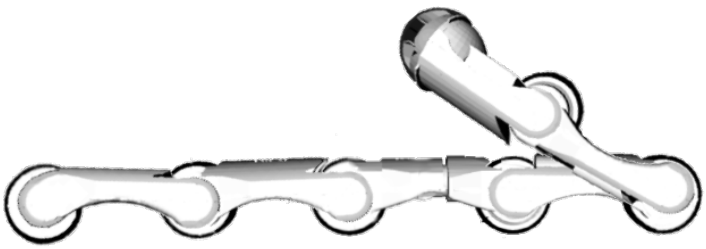
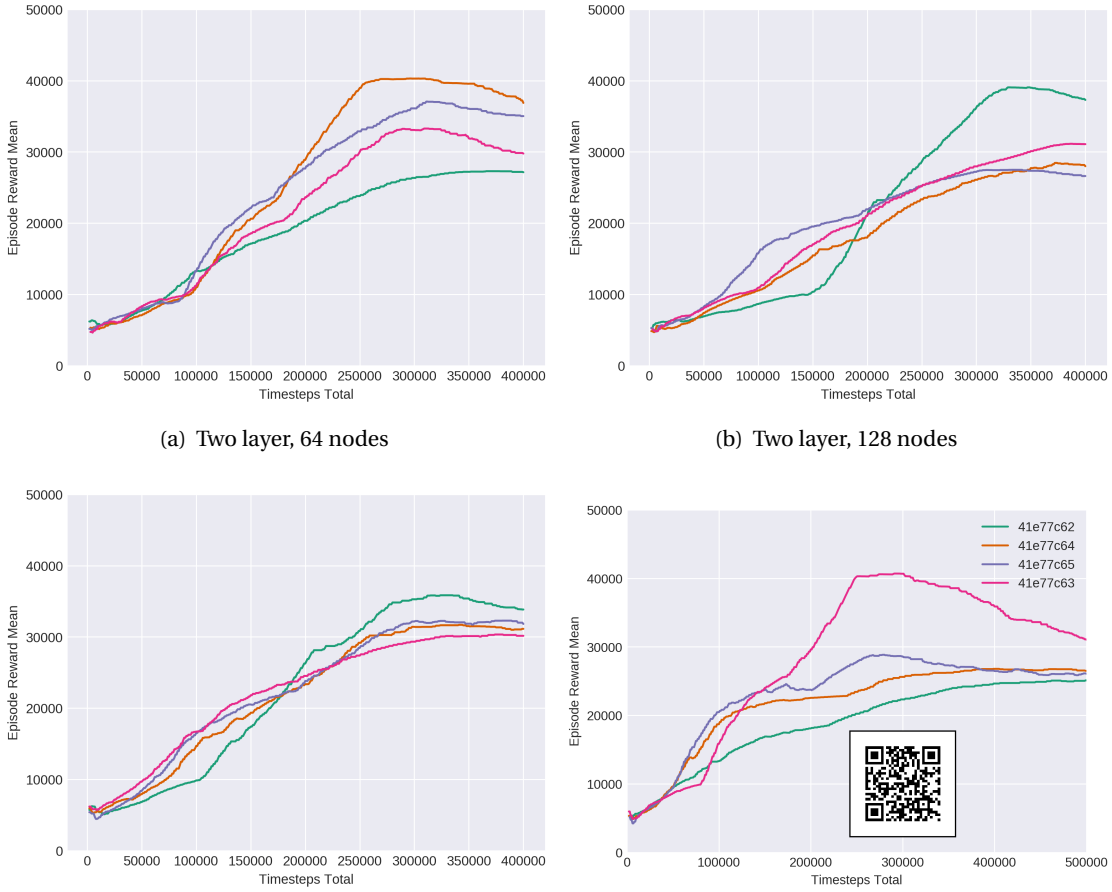


Figure A.1: Joint Problem

B Appendix 2



(a) Two layer, 64 nodes

(b) Two layer, 128 nodes

(c) Three layer, 64 nodes

(d) Three layer, 128 nodes

Figure B.1: Experiment: different layer and nodes.

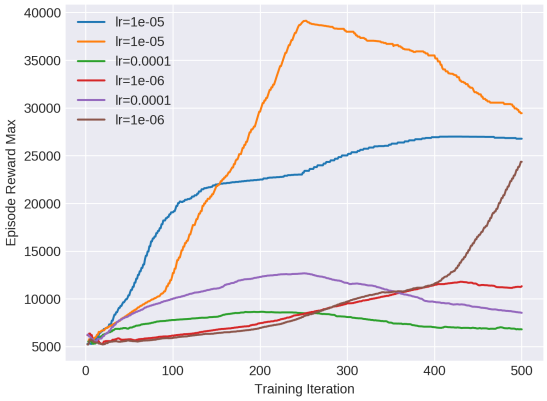
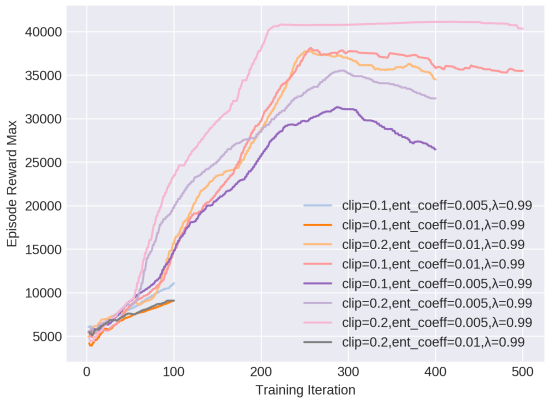
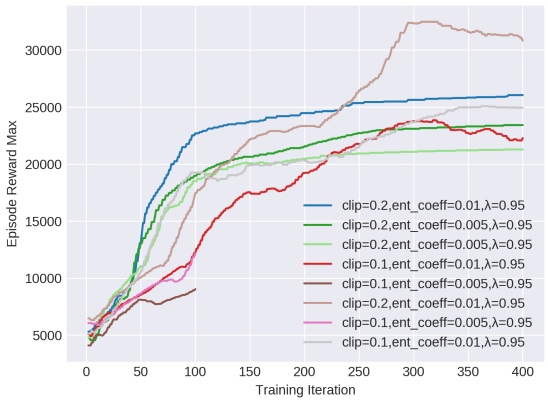


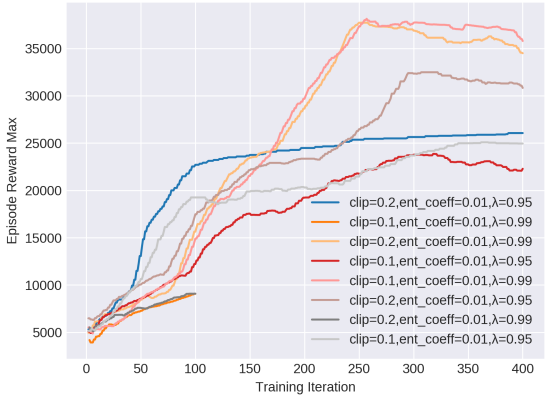
Figure B.2: Results: Grid search of learning rate.



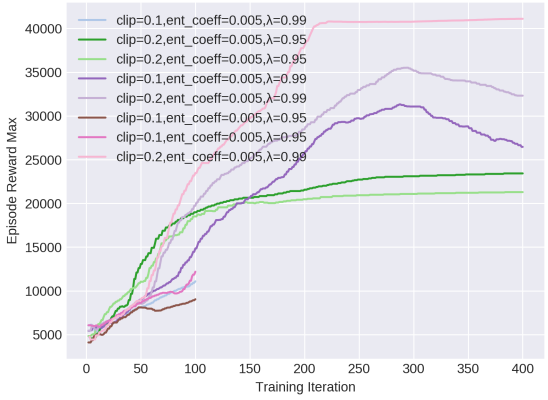
(a) lambda=0.99



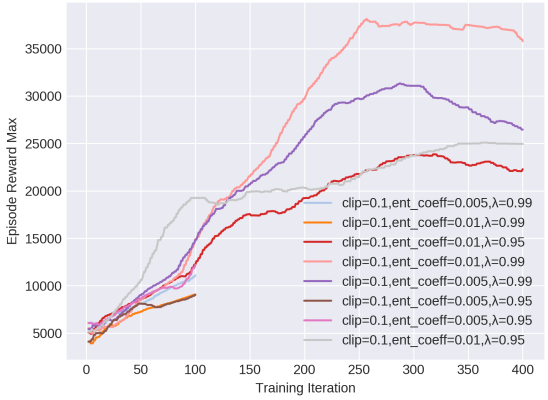
(b) lambda=0.95



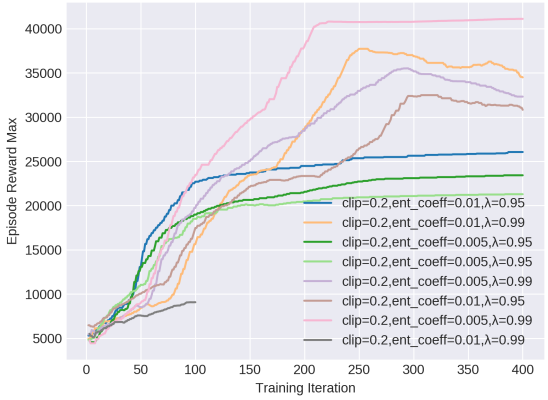
(c) ent_coeff=0.01



(d) ent_coeff=0.005



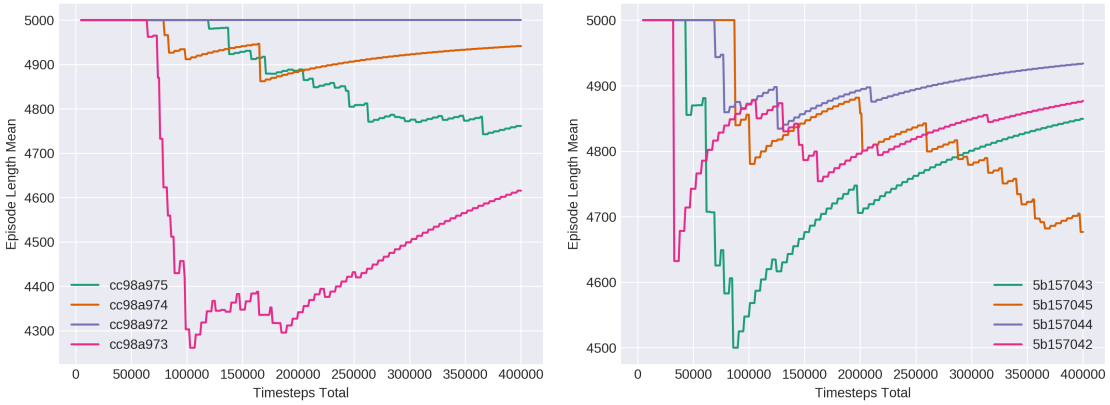
(e) clip=0.1



(f) clip=0.2

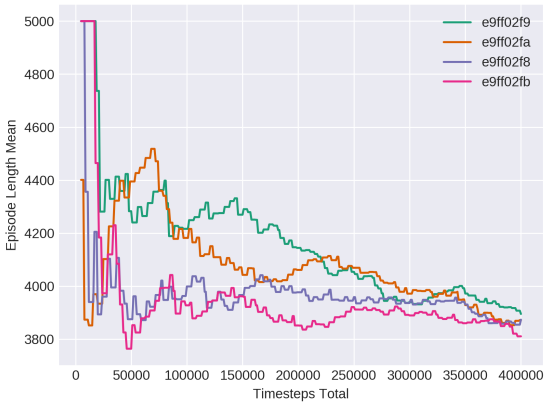
(g)

Figure B.3: Results of the grid-search for lambda, entropy coefficient and clip parameter. The 16 experiments are plotted in 6 figures. Each figure shows experiments which have 1 mutual grid-search parameter. A YouTube video of the best performing experiment (pink, clip=0.2,ent_coeff=0.005,λ=0.99) can be watched by scanning the QR code.

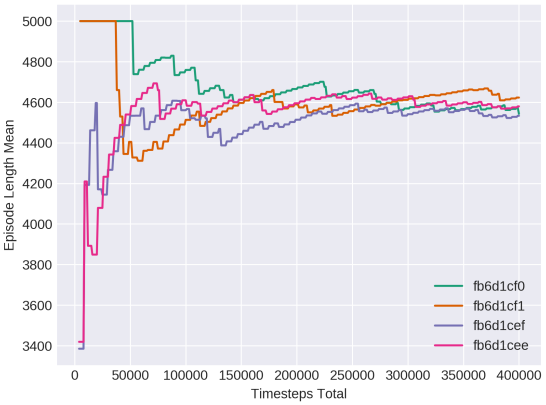


(a) SRL visual.

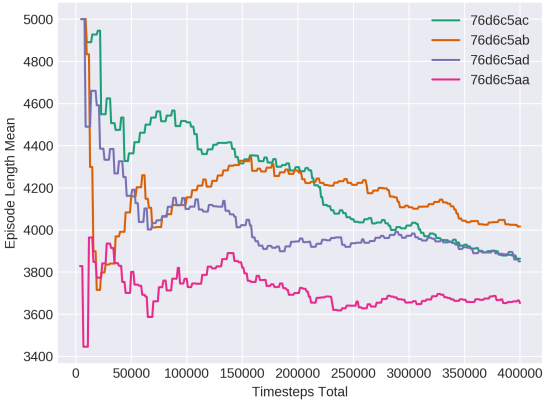
(b) SRL LSTM



(c) HRL Vision

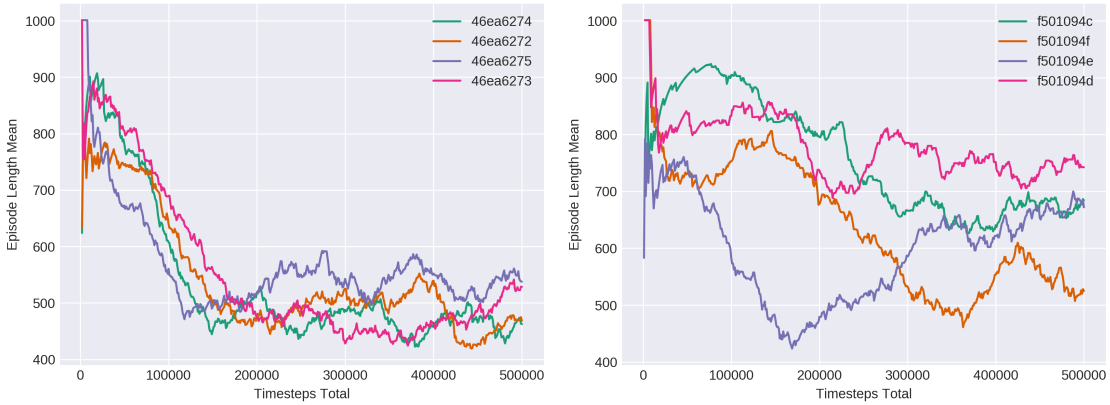


(d) HRL Simul



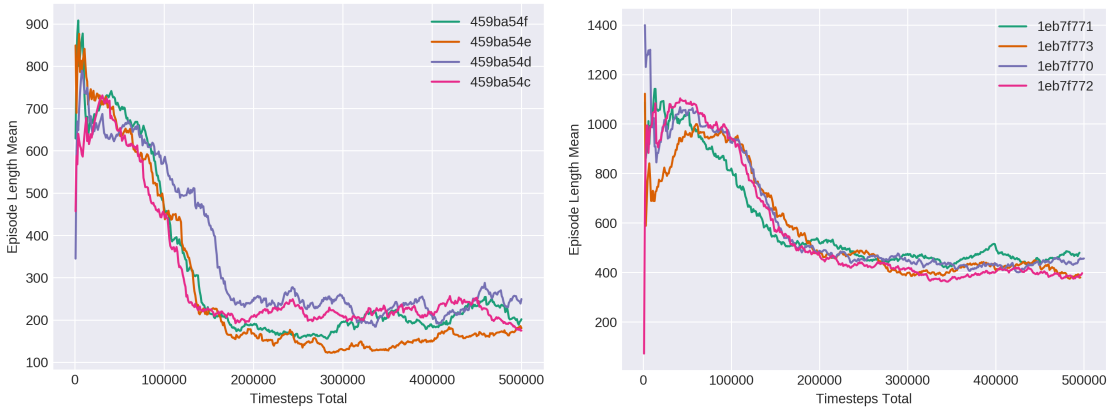
(e) HRL LSTM

Figure B.4: Average episode lengths of the sharp SRL and HRL experiments.



(a) Enter-turn Policy.

(b) Enter-Turn Policy with LSTM.



(c) Outward-Turn Policy.

(d) Clamp-Drive Policy.

Figure B.5: Separate training of sub-policies: Average episode lengths.

C Appendix 3

C.1 Running Ray with V-REP

These instructions will get you a copy of the project up and running on your local machine.

C.1.1 Prerequisites

- V-REP 3.6.2
- Python 3.6
- PyCharm

C.1.2 Installing

Pull the project

```
git clone --recurse-submodules
git@github.com:Luukgr/pirate-vrep-rl.git
```

Set environment variables in /.bashrc. Set the V-Rep install Dir between <>

```
export VREP_ROOT=<EDIT/ME/PATH/TO/VREP_ROOT/INSTALL/DIR>
export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$COPPELIASIM_ROOT
export QT_QPA_PLATFORM_PLUGIN_PATH=\$VREP_ROOT
```

Start PyCharm by command line. See

Make virtualenv

```
source venv/bin/activate
```

Install requirements.txt

```
pip3 install -r requirements.txt
```

Install Pyrep

```
cd PyRep
pip3 install -r requirements.txt
python3 setup.py install
```

Install Pirate Gym Environment:

```
pip3 install -e .
```

Bibliography

Abdellatif, M., H. Mohamed, M. Hesham, A. Abdelmoneim, A. Kamal and A. Khaled (2018), Mechatronics Design of an Autonomous Pipe-Inspection Robot, in *MATEC Web Conf. Volume 153*.

Adria, O., H. Streich and J. Hertzberg (2004), Dynamic Replanning in Uncertain Environments for a Sewer Inspection Robot, **vol. 1**, no.1, p. 4, ISSN 1729-8814, 1729-8814, doi:10.5772/5617.
http://journals.sagepub.com/doi/10.5772/5617

Anyscale (2020), RLlib: Scalable Reinforcement Learning — Ray 0.9.0.dev0 documentation.
https://ray.readthedocs.io/en/latest/rllib.html

AurelianTactics (2018), PPO Hyperparameters and Ranges, library Catalog: medium.com.
https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe

Barbero, M. (2018), Reinforcement learning for robot navigation in constrained environments.
http://essay.utwente.nl/76385/

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba (2016), OpenAI Gym.

Choi, H. R. and S. M. Ryew (2002), Robotic system with active steering capability for internal inspection of urban gas pipelines, in *Mechatronics Volume 12*.

CoppeliaRobotics (2019a), Building a clean model tutorial.
https://www.coppeliarobotics.com/helpFiles/en/buildingAModelTutorial.htm

CoppeliaRobotics (2019b), Designing dynamic simulations.
https://www.coppeliarobotics.com/helpFiles/en/designingDynamicSimulations.htm

Dayan, P. and G. E. Hinton (1993), Feudal Reinforcement Learning.
https://www.cs.toronto.edu/~fritz/absps/dh93.pdf

Debenest, P., M. Guarnieri and S. Hirose (2014), PipeTron series - Robots for pipe inspection, *Proceedings of the 2014 3rd International Conference on Applied Robotics for the Power Industry*, pp. 1–6.

Dertien, E. (2014), *Design of an inspection robot for small diameter gas distribution mains*, Ph.D. thesis, University of Twente, Netherlands, doi:10.3990/1.9789036536813.

Espeholt, L., H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg and K. Kavukcuoglu (2018), IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures, *arXiv:1802.01561 [cs]*, arXiv: 1802.01561.
http://arxiv.org/abs/1802.01561

Fei-Fei, L., J. Justin and Y. Serena (2017), Lecture Collection | Convolutional Neural Networks for Visual Recognition (Spring 2017).
https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

Folkman, S., J. Rice, A. Sorenson and N. Braithwaite (2012), Survey of Water Main Failures in the United States and Canada, *AWWA Journal*, **vol. 104**, pp. 70–79, doi:10.5942/jawwa.2012.104.0135.

Franceschetti, A., E. Tosello, N. Castaman and S. Ghidoni (2018), Robotic Arm Control and Task Training through Deep Reinforcement Learning.

Frans, K., J. Ho, X. Chen, P. Abbeel and J. Schulman (2017), Meta Learning Shared Hierarchies, *arXiv:1710.09767 [cs]*, arXiv: 1710.09767.
http://arxiv.org/abs/1710.09767

Garza Morales, G. (2016), Increasing the Autonomy of the Pipe Inspection Robot PIRATE.
http://essay.utwente.nl/71300/

Geerlings, N. (2018), Design of a high-level control layer and wheel contact estimation and compensation for the pipe inspection robot PIRATE.
https://repository.tudelft.nl/islandora/object/uuid:0db67388-371b-4f54-bb3c-b07521043637?collection=education

Golovin, D., B. Solnik, S. Moitra, G. Kochanski, J. Karro and D. Sculley (2017), Google Vizier: A Service for Black-Box Optimization, in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Association for Computing Machinery, Halifax, NS, Canada, KDD '17, pp. 1487–1495, ISBN 978-1-4503-4887-4, doi:10.1145/3097983.3098043.
https://doi.org/10.1145/3097983.3098043

Hausknecht, M. and P. Stone (2017), Deep Recurrent Q-Learning for Partially Observable MDPs, *arXiv:1507.06527 [cs]*, arXiv: 1507.06527.
http://arxiv.org/abs/1507.06527

Henderson, P., R. Islam, P. Bachman, J. Pineau, D. Precup and D. Meger (2019), Deep Reinforcement Learning that Matters, *arXiv:1709.06560 [cs, stat]*, arXiv: 1709.06560.
http://arxiv.org/abs/1709.06560

Hui, J. (2018), RL — Proximal Policy Optimization (PPO) Explained, library Catalog: medium.com.
https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12

Isbell, C. and M. Littman (2016), RL Course by Udacity.
https://classroom.udacity.com/courses/ud600

Islam, R., P. Henderson, M. Gomrokchi and D. Precup (2017), Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control, *arXiv:1708.04133 [cs]*, arXiv: 1708.04133.
http://arxiv.org/abs/1708.04133

Jaderberg, M., V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando and K. Kavukcuoglu (2017), Population Based Training of Neural Networks, *ArXiv*, **vol. abs/1711.09846**.

James, S., M. Freese and A. J. Davison (2019), PyRep: Bringing V-REP to Deep Robot Learning, *arXiv:1906.11176 [cs]*, arXiv: 1906.11176.
http://arxiv.org/abs/1906.11176

Kumar, S. (2019), Development of SLAM algorithm for a Pipe Inspection Serpentine Robot.
http://essay.utwente.nl/80207/

Larochelle, H., Y. Bengio, J. Louradour and P. Lamblin (2009), Exploring Strategies for Training Deep Neural Networks, *J. Mach. Learn. Res.*, **vol. 10**, pp. 1–40.

Li, L., K. G. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht and A. Talwalkar (2018), Massively Parallel Hyperparameter Tuning, *ArXiv*, **vol. abs/1810.05934**.

Liaw, R., E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez and I. Stoica (2018), Tune: A Research Platform for Distributed Model Selection and Training, *arXiv:1807.05118 [cs, stat]*, arXiv: 1807.05118.
http://arxiv.org/abs/1807.05118

Masters, D. and C. Luschi (2018), Revisiting Small Batch Training for Deep Neural Networks, *arXiv:1804.07612 [cs, stat]*, arXiv: 1804.07612.
http://arxiv.org/abs/1804.07612

Meyer, D. (2016), Notes on policy gradients and the log derivative trick for reinforcement learning.
http://www.1-4-5.net/~dmm/ml/log_derivative_trick.pdf

Moritz, P., R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan and I. Stoica (2018), Ray: A Distributed Framework for Emerging AI Applications, *arXiv:1712.05889 [cs, stat]*, arXiv: 1712.05889.
http://arxiv.org/abs/1712.05889

Murdoch, E. (2012), *The Standard, A Masters's Guide To Ships' Piping*, CHARLES TAYLOR & CO. LIMITED.

Nassiraei, A. A. F., Y. Kawamura, A. Ahrary, Y. Mikuriya and K. Ishii (2007), Concept and Design of A Fully Autonomous Sewer Pipe Inspection Mobile Robot "KANTARO", *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 136–143.

Parvez, J. (2018), Life Cycle Assessment of PVC Water and Sewer Pipe and Comparative Sustainability Analysis of Pipe Materials, **vol. 2018**, no.7, pp. 5493–5518, ISSN 1938-6478, doi:10.2175/193864718825138925.
https://accesswater.org/publications/-300020/life-cycle-assessment-of-pvc-water-and-sewer-pipe-and-comparative-sustainabili

Rohmer, E., S. P. N. Singh and M. Freese (2013), CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework, in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, www.coppeliarobotics.com.

Rome, E., J. Hertzberg, F. Kirchner, U. Licht and T. Christaller (1999), Towards autonomous sewer robots: the MAKRO project, **vol. 1**, no.1, pp. 57–70, ISSN 1462-0758, doi:10.1016/S1462-0758(99)00012-6.
http://www.sciencedirect.com/science/article/pii/S1462075899000126

Rufus, N., U. K. R. Nair, A. V. S. S. B. Kumar, V. Madiraju and K. M. Krishna (2020), SROM: Simple Real-time Odometry and Mapping using LiDAR data for Autonomous Vehicles, *arXiv:2005.02042 [cs]*, arXiv: 2005.02042.
http://arxiv.org/abs/2005.02042

Schulman, J., S. Levine, P. Moritz, M. I. Jordan and P. Abbeel (2017a), Trust Region Policy Optimization, *arXiv:1502.05477 [cs]*, arXiv: 1502.05477.
http://arxiv.org/abs/1502.05477

Schulman, J., P. Moritz, S. Levine, M. Jordan and P. Abbeel (2018), High-Dimensional Continuous Control Using Generalized Advantage Estimation, *arXiv:1506.02438 [cs]*, arXiv: 1506.02438.
http://arxiv.org/abs/1506.02438

Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov (2017b), Proximal Policy Optimization Algorithms, *arXiv:1707.06347 [cs]*, arXiv: 1707.06347.
http://arxiv.org/abs/1707.06347

Selvarajan, A., A. Kumar, D. Sethu and M. A. bin Ramlan (2019), Design and Development of a Snake-Robot for Pipeline Inspection, *2019 IEEE Student Conference on Research and Development (SCOReD)*, pp. 237–242.

Silver, D. (2015), RL Course by David Silver.
https://www.youtube.com/watch?v=2pWv7GOvuf0&t=3800s

Sutton, R. S. and A. G. Barto (2018), *Reinforcement learning: an introduction*, Adaptive computation and machine learning series, The MIT Press, Cambridge, Massachusetts, second edition edition, ISBN 978-0-262-03924-6.

Sutton, R. S., D. Precup and S. Singh (1999), Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning, **vol. 112**, no.1, pp. 181–211, ISSN 0004-3702, doi:10.1016/S0004-3702(99)00052-1.
http://www.sciencedirect.com/science/article/pii/S0004370299000521

test-equipment.com (2019), test-equipment.com.
https://www.test-equipment.com.au/drain-sewer-pipe-inspection-cctv-camera-system-with-120m-reel-50mm-ccd-came

VJ TECHNOLOGIES (2013), Digital X-ray Solutions for Oil & Gas X-ray Inspection.
https://vjt.com/wp-content/uploads/2014/02/OilGas-Rev-2a.pdf

WILLIAMS, R. J. and J. PENG (1991), Function Optimization using Connectionist Reinforcement Learning Algorithms, **vol. 3**, no.3, pp. 241–268, ISSN 0954-0091, doi:10.1080/09540099108946587, publisher: Taylor & Francis _eprint: https://doi.org/10.1080/09540099108946587.
https://doi.org/10.1080/09540099108946587

Zeng, X. (2019), Reinforcement learning based approach for the navigation of a pipe-inspection robot at sharp pipe corners.
http://essay.utwente.nl/79790/