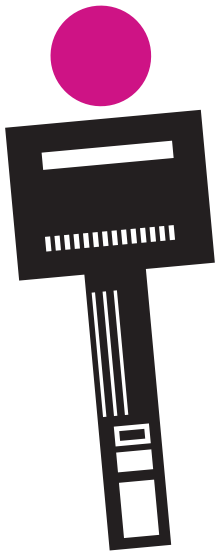


RAM ROBOTICS AND MECHATRONICS

PIRATE ROBOT AUTONOMOUS NAVIGATION THROUGH COMPLEX PIPE NETWORKS USING REINFORCEMENT LEARNING

P. (Paul) Vacariu

MSC ASSIGNMENT

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

September, 2021

064RaM2021
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

Summary

There are many industries that rely on pipe systems. Regardless of what they transport, a leak caused by pipe failure can prove to be either costly, disastrous, or even both. Therefore, it is important to perform periodic inspections. In some cases, especially when the pipes are easily accessible, this task can be achieved in a relatively easy way. Unfortunately this is not always the case, for instance in case of pipes buried underground. In such situations the maintenance becomes expensive, in both time and resources. The solution comes in the form of camera systems that can be used to inspect the pipes. Out of all the options, the autonomous pipe inspections robots have the most advantages, as they have the possibility to travel long distances, navigate through bends and they do not require too much human intervention.

This project focuses on improving the navigation controller for the Pipe Inspection Robot for AuTonomous Exploration (PIRATE). The robot itself employs several modular sections separated by rotational joints and is equipped with wheels that allow it to navigate through pipe systems of various diameters. The autonomous navigation uses a Hierarchical Reinforcement Learning architecture, consisting of multiple agents organized in a hierarchical way. The top level agent employs the options framework method to pick one of the subpolicies, which then generate actions sent to the robot.

In this project, the hierarchical model is extended. The goal is to introduce a decision mechanism that allows the PIRATE robot to pick one of the bends in a T-junction and navigate through it. To achieve this goal, the hierarchical structure is extended by complementing it with a feudal learning approach. Thus, the top level policy generates not only an option regarding picking one of the low level policies, but also a subgoal that is passed to the active subpolicy and acts as a decisional element.

Being able to navigate through T-junctions opens up the possibility to perform certain inspection missions. In case the pipe network is already known, the robot can trace a predetermined path to reach a certain position. If there is no previous information related to the structure of the pipe system, the robot can explore it and gather data which can be used to build a virtual map.

One extra aspect studied in this project is hyperparameter tuning. Using a Population Based Training approach, the parameters of each policy can be tuned, thus increasing the performance and robustness of each agent.

Contents

1 Introduction

1.1 Context

Extensive pipe systems are used for various applications: sewage, water and gas distribution, factories, and other industrial applications. In time, the pipes wear out and maintenance has to be performed on them. However, the task is often not trivial, due to the pipe network being buried underground, or several meters above ground. Reaching them for manual inspections becomes a time and resource consuming task. Therefore, employment of smart tools is needed to reduce the costs of maintenance.

Another aspect to consider is the life expectancy of the pipes. Performing maintenance too early causes a greater increase in costs, both in terms of maintenance and in terms of operation, as the pipe operation needs to be stopped. On the other hand, waiting too long for the next inspection can lead to pipes failing. Therefore, it is important to know when is the right time to perform inspections. Unfortunately the life expectancy of pipes varies greatly due to environment conditions, materials, the flowing speed and the substances that pass through the pipes Utanohara and Murase (2019).

To overcome these challenges, it is possible to perform maintenance from the inside of the pipes instead. In this way, the difficulties of manual inspection of the outside of the pipes are mitigated. One device that can be used for this task is called a PIG (Pipe Inspection Gauge). They are inserted in one part of the pipe and are retrieved in another point downstream. Traditionally, the PIG is used for cleaning the inside of the pipes, but more intelligent variants equipped with sensors, are able to inspect the conditions of the pipe walls.

Another solution to the pipe inspection problem comes in the form of tools equipped with a camera or other types of sensors and a long cable attached to them. These devices are pushed through the pipe and record images of the walls. By employing regular checks, possible damage to the pipes can be exposed way before the pipes break down. The main drawback of this system is that it does not take into consideration bends and corners in the network. Therefore, they cannot be used in complex piping systems.

Autonomous robots can solve the issue of navigating complex pipe networks. These machines are able to navigate through pipe networks without human intervention and map the areas they visit. Processing the data they collect either online or offline, the status of the pipes can be assessed, and the location of the possible damage can be identified. One example of such automatons is the Pipe Inspection Robot for AuTonomous Exploration (PIRATE), presented in Dertien (2014). The robot is able to rotate and bend in order to fit and turn even in T-shaped pipe intersections. It can also fit through pipes of varying diameters because of its propulsion method: the robot clamps its modules in order to touch the opposing walls. The front module can be equipped with different types of sensors, including a camera used both for navigation and for the actual pipe inspection task. A schematic representation of the robot is shown in Figure 1.1.

1.2 Problem Statement

The approach considered in this project uses Reinforcement Learning (RL) for learning a navigation strategy for the PIRATE robot. The navigation controller chooses a setpoint value for the motor control taking into account the sensory readings from the robot.

Learning the optimal strategy with RL on the real robot is far from being an easy task. On each new training episode, the position and pose of the robot needs to be manually reset. Additionally, collecting the amount of data needed for learning a robust model from the interaction

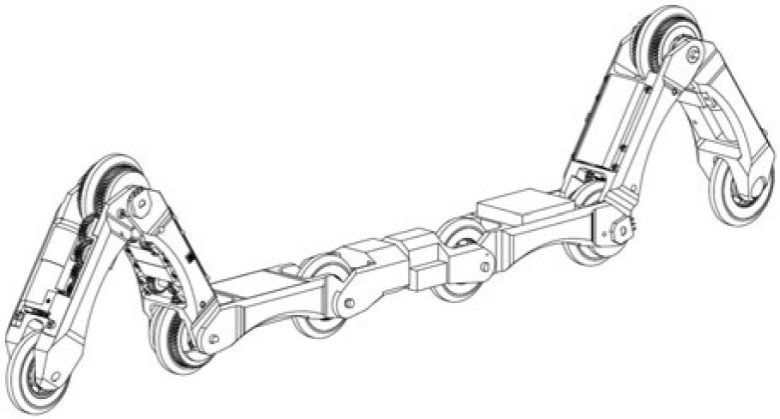


Figure 1.1: Schematic representation of the PIRATE robot. Image from Dertien (2014)

between the robot and the environment would take a lot of time. Thus, it is preferred to firstly train an initial navigation controller inside a simulation environment, by using the dynamic model of the robot. The resulting RL model obtained after this training can then be hopefully transferred on the real system.

The work of Grefte (2020), on which this project is based, proposes a Hierarchical Reinforcement Learning (HRL) control model for the navigation of the PIRATE robot in a navigation environment through straight pipes and 90 degrees turns.

The aim of this project is to improve upon the existing research. Firstly, the current model is adapted to allow navigation through T-junctions and pipe diameter changes. Secondly, the robustness of the model needs to be increased, such that the resulting navigation controller can be moved from a simulation environment on the real robot.

1.3 Research Questions

- How can the current HRL framework be adapted to support pipe diameter changes and navigation through T-junctions?
- What decision model for T-junctions can be used to ensure full exploration of the pipe network?
- How can the HRL model be extended to support different types of inspection missions (e.g. exploring a branching pipe network, following predefined paths)?
- Can a population based training approach be used to increase the robustness of the PIRATE navigation controller model, compared to the present implementation?

1.4 Outline

Chapter 2 contains the theoretical background information about the tools and technologies used in this research. This includes a short description about RL and the construction of the PIRATE robot. The related work to this research is described in Chapter 3. Afterwards, Chapter 4 presents the methods used in this project to solve the research questions posed in Section 1.3. The experiments performed require certain preparations, which are described in Chapter 5. The results of the newly implemented methods are presented in Chapter 6, which are then analysed and discussed in the same chapter. Finally, a conclusion to the whole project is formulated in Chapter 7.

2 Background

This chapter presents background information needed to understand the concepts presented in the project. First, a general overview about Machine Learning is presented. Then, the focus shifts on Reinforcement Learning and Hierarchical Reinforcement Learning. Finally, background information related to the PIRATE robot is shown.

2.1 Machine Learning

Machine Learning encompasses different types of algorithms that, using an initial dataset and a parametrized mathematical model, computes the parameters of the model such that a predefined cost function is minimized. The resulting model is able to make predictions or decisions based on the initial data without the need of specifying any ruleset. Depending on the type of data provided to the learning algorithm, there are three main types of approaches: supervised learning, unsupervised learning and reinforcement learning. All of the methods have in common that fitting the model implies minimizing a cost function, which depends on the parameters of the model.

In supervised learning, the data used for training contains labels, representing the ground truth output for every input combinations provided. By using this method, a linear or nonlinear function is fitted using the training data. Running the trained model with a different input makes a prediction of the output. For example, supervised learning can be used to calculate the expected salary per month with respect to the number of years of experience. Starting from existing data, namely information about the current employees, a model is trained, and later, when a new employee is hired, his salary can be computed based on his experience.

On the other hand, unsupervised learning does not use labelled data. Instead, the aim of the algorithms to learn patterns from the provided data. For instance, identifying different types of objects in an image can be solved as a clustering problem, using unsupervised learning to group similar pixels together (Kim et al., 2020). It should be noted that this method would only result in finding the boundaries between the objects, and not classifying the obtained clusters.

Finally, reinforcement learning uses data obtained from the interaction of an agent in an environment. The agent has a specific goal, and can perform different types of actions, while also receiving information about the environment. Depending on its current state and the action chosen, the agent receives a reward. The purpose of the training algorithm is to maximize the obtained reward, thus deciding which action is better to pick in a certain scenario. A robot controller learning to navigate an unknown environment is a common application for this subclass of ML algorithms.

Every subclass of ML algorithms has different applications and use cases. In this project, RL is the main focus, and different aspects of the topic are described in depth in the following sections.

2.2 Reinforcement Learning

The majority of RL problems can be modelled using a Markov Decision Process (MDP). This section introduces the concept of an MDP and the tools needed to solve it.

2.2.1 Markov Decision Process

An MDP is a mathematical framework for modelling decision makin process in presence of uncertainties. It is defined by a set of states, S, also called state space; a set of actions, A, also called the action space; a reward function $R : SxAxS \rightarrow \mathbb{R}$, obtained from transitioning from state s to

state s' by taking action a; and a state transition model, $P : SxAxS \rightarrow [0,1]$, representing the probability of reaching state s' from state s by taking action a.

One interesting property that the MDP possesses is that the probabilities of reaching future states is independent on the past states - it only depends on the current state. This is also called the Markov property and its mathematical form is presented in (2.1).

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2, ..., S_t) \tag{2.1}$$

A policy is defined as a probability distribution function, $\pi : SxA \rightarrow [0,1]$, that maps a state and an action to the probability that action a is picked from state s (2.2).

$$\pi(s,a) = P(a|s) \tag{2.2}$$

The basic RL problem is to find a policy that maximizes the cumulative discounted reward G, also called the return, obtained by running the MDP, over multiple trials, starting from a particular state, s_0. Equation (2.3) shows how this reward is calculated. Here $\gamma \in [0,1]$ represents the discount factor, and it used to represent the fact that future reward is less important than the present reward for each time step. If γ is equal to 1, the future rewards are not discounted, and the returns represents the total rewards earned in the current episode. If γ is equal to 0, then the future reward is discarded, and the returns becomes equal to the reward obtained in the current state transition.

$$G = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \tag{2.3}$$

In case of a limited run of the MDP (for instance, in situations where there is an end state), the episode contains a finite number of state transitions. Therefore, the return will have a finite number of terms as well.

2.2.2 Value Functions and the Bellman Equation

It is important to assess how good it is for the RL agent to be in a particular state. One measure of optimality is the expected return starting from a state s and following the policy π. This function is also called the state-value function (2.4). The purpose of RL can therefore be formulated as finding the policy that maximizes the state-value function. This is also called an optimal policy and is denoted by π^*.

$$V_\pi(s) = \mathbb{E}[G|s, \pi] \tag{2.4}$$

Besides the state-value function, there is a metric that evaluates how rewarding is picking a certain action in a specific state. This is called the action-value function Q, also known as the Q-function, and it is defined as the expected return starting from a particular state s and taking action a (2.5). It can be noted that the action-value function still depends on the chosen policy, because the future rewards depend on the future actions taken. The state-value function can be expressed as a weighted sum of the action-value function over the action space, using the policy's probabilities as weights (2.6).

$$Q_\pi(s,a) = \mathbb{E}[G|s, a, \pi] \tag{2.5}$$

$$V_\pi(s) = \sum_a \pi(s,a) \cdot Q_\pi(s,a) \tag{2.6}$$

The value functions can be computed in multiple ways. One option is to use a recursive procedure based on the Bellman Equation (2.7) to calculate the value function. The equation is a weighted sum over the possible state transitions using the policy's probability as the weights. Each term has two components: the immediate reward obtained from the state transition, $R(s,a)$, and the discounted value function of the next state $V_\pi(s')$.

$$V_\pi(s) = \sum_a \pi(s,a) \cdot \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_\pi(s')) \tag{2.7}$$

Substituting the state-value function with the action-value function using (2.6) into the Bellman Equation (2.7), the Bellman equation for the action-value function is obtained (2.8).

$$Q_\pi(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \sum_{a'} \pi(s',a') \cdot Q_\pi(s',a')) \tag{2.8}$$

2.3 Dynamic Programming

As mentioned above, the purpose of an RL agent is to find the policy that maximizes the value functions. The Bellman equation can be applied to the optimal state-value and action-value function, resulting in the forms presented in (2.9) and (2.10), also called the Bellman optimality equation.

$$V_*(s) = \max_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_*(s')) \tag{2.9}$$

$$Q_*(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q_*(s',a')) \tag{2.10}$$

The optimum policy can be extracted from the optimal value function by finding the action that maximizes the value function. The result is a deterministic greedy policy. The equation is shown in (2.11).

$$\pi_*(s) = argmax_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_*(s')) \tag{2.11}$$

These equations are used to determine the optimal policy. The next subsections show different methods that can be employed to achieve this goal. In this section, it is presumed that the state transition model and the reward fucntion of the MDP are known.

2.3.1 Policy Iteration

The Policy Iteration algorithm alternates between a policy evaluation and a policy improvement step. During the policy evaluation, the Bellman equation in the form of (2.7) is used to update the value of the state-value function for each state in the state space. This evaluation continues until the values of the value function do not change that much. The second step, the policy improvement step, updates the policy to the greedy policy that causes the agent to take the action that maximizes the value function for the current state - see also (2.11). If the second step caused an update in the policy, this means that the algorithm did not yet reach the optimal policy, and the next iteration continues with the current estimates for the value function and the policy. Otherwise, the obtained policy is the optimal optimal one, and the algorithm stops.

Figure 2.1 shows a schematic representation of the algorithm. Starting from an initial guess of the policy and value function, each iteration updates them and, eventually, converges to the optimal policy.

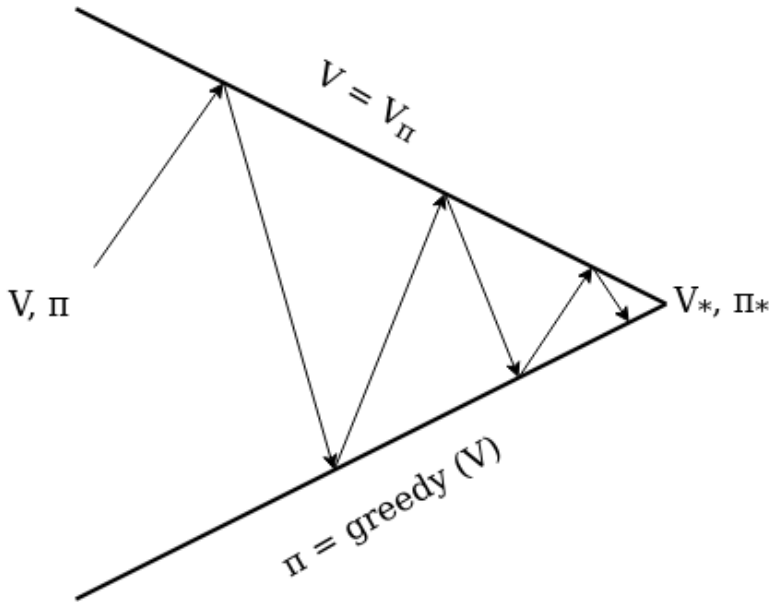


Figure 2.1: The policy iteration algorithm

2.3.2 Value Iteration

In policy iteration, each step is performed all the way: the policy evaluation step stops only when the value function reaches its true value, and the policy gets updated to the full greedy policy. In the value iteration approach, the two steps are merged together. There is no full value function evaluation or full policy update. The algorithm uses the Bellman Optimality Equation (2.9) as an update rule for the value function. When the maximum value change over all the state of the state-value function becomes smaller than a set threshold, the optimal value function is reached, and the optimal policy is computed using (2.11).

2.4 Reinforcement Learning Sampling Methods

Unfortunately, having complete knowledge of the underlying MDP is not always possible. If the model is too complex, an accurate state transition, state space or reward model can be hard to deduce. Therefore, different methods need to be employed to calculate the value function in the policy evaluation step. One solution is to use experience gained from running the agent in the environment: the rewards gained from each step of the simulation are used to update the current estimate of the value function for the appropriate state.

When running the agent in the environment to sample new data, the next state is reached by taking an action dictated by a policy. There are two approaches for this: either the policy used for sampling is the same as the one used in the improvement step (also called on-policy learning), or it is a different one (called off-policy learning). It should be noted that if the policy used for training is purely greedy, the agent may never learn new strategies. It is possible that a non-greedy initial decision yields a higher payoff on the long run, by accruing more reward in the future steps. Therefore, a greedy policy may not yield the optimum result.

Assuming the action space is discrete, a good candidate for an alternative non-greedy policy is the $\epsilon - greedy$ policy. In this policy, the agent performs the greedy action with a probability equal to $1-\epsilon$, where ϵ is a number between 0 and 1. In the rest of the cases, the agent picks a random action from the possible action space at its current state. Using this type of policy allows the agent to explore new options in the environment, while at the same time exploiting known knowledge to maximize its cumulative reward.

In this section two sampling methods are explored, namely Monte Carlo and Temporal Difference (TD). Furthermore, the differences between on-policy and off-policy learning are explained.

2.4.1 Monte Carlo Learning

An episode is a sequence of states and actions that ends with a terminal state. In Monte Carlo learning, the value function for each state is evaluated by letting the agent run in the environment for a whole episode, starting from that particular state, and taking the mean of the return for all the episodes. The return is calculated as a sum of discounted rewards, according to (2.3). Over multiple iterations, an estimated value function is obtained, which can then be used as the result of the evaluation step of the policy iteration algorithm.

In an alternative approach, instead of keeping track of the whole return over multiple episodes, the estimated value function can be updated using its previous value and the error between the latest return and the current value. This can be seen in (2.12), where $V(s)$ is the estimate of the value function at state s, $N(s)$ is the number of episodes ran from state s, and G is the return for the current episode.

$$V(s) \leftarrow V(s) + \frac{1}{N(s)}(G - V(s)) \tag{2.12}$$

Unfortunately the classical Monte Carlo approach can only be used for episodic tasks. In other words, if the underlying MDP has no definite end state, a total return cannot be computed from observations alone. Furthermore, in some cases episodes may take a long time to finish, and thus evaluating the value function becomes a slow process.

2.4.2 Temporal Difference Learning

In TD learning, the value function is updated after each step. The agent takes an action using the current policy, receives a reward, and this value is used, alongside the estimated value function for the next time, to update the value function at the current state. The update rule uses (2.13), where $V(s)$ is the value function at state s, R is the gained reward, s' is the next state, $N(s)$ represents the number of times the current state has been visited, and γ is the discount factor. The term in the parenthesis is also called the TD error.

$$V(s) \leftarrow V(s) + \frac{1}{N(s)}(R + \gamma V(s') - V(s)) \tag{2.13}$$

This method can be adapted to any number of steps. Instead of updating the value function using the value function of the next step, multiple state transitions can be considered. It is noted that if the number of steps because high enough, the algorithm transforms into the Monte Carlo approach. In contrast with the Monte Carlo approach, the algorithm needs an initial estimate for the value function. This introduces a bias in the estimated value of the value function, which decreases as the number of evaluations increases.

Furthermore, it is not necessary to scale the TD error by the count of the visits of the state. Using a fixed step size is a valid alternative for the update rule (2.13).

2.4.3 On-Policy Learning

In on-policy learning approaches, the policy used for taking information from the environment in the policy evaluation step is then getting improved in the policy improvement step. In other words, the aim is to improve the policy that makes the decisions.

Example methods that use on-policy leaning are SARSA and Proximal Policy Optimization. They both use the TD learning approach for sampling the environment.

2.4.4 Off-Policy Learning

For the case of off-policy learning approaches, estimating the value function does not take into account the policy being trained. This means that the evaluation step uses a different policy than the one used by the agent. Q-Learning represents one example algorithm that uses off-policy learning.

2.4.5 Gradient Methods

Policy gradient methods represent a family of RL techniques that use gradient descent to optimize the parameters of a policy, with respect to the expected return. They can be used for model-free or model-based approaches, but are in general on-policy. An issue with classical gradient descent methods is that they are sensitive to the step size. Using the natural gradient instead solves this issue, however the natural gradient involves calculation of second-order derivative matrix, which is not feasible for large scale problems.

2.4.6 Example RL Algorithms

In this section, a few RL algorithms are presented. All of them use sampling methods, and differ by whether they use on-policy, off-policy methods, and in the way the value function gets updated each iteration.

SARSA

SARSA stands for state-action-reward-state-action. The approaches presented so far focus on learning the value of the state using state transitions. In SARSA, the state-actions pairs are used to learn the action-value function for each pair. The update rule is presented in (2.14). Here, $Q(s,a)$ is the action-value function at state s taking action a, s' is the state reached by taking action a from s, decided using a policy derived from Q (for instance, ϵ-greedy), a' is the action chosen from state s' using the same policy, and α is the fixed step size for the update.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma Q(s',a') - Q(s,a)) \tag{2.14}$$

The policy is updated consistently after each update of the action-value function.

Actor-Critic Architecture

Actor-Critic methods separate completely the representation of the policy from the representation of the value function. An actor uses the policy to generate an action from the current state, which interacts with the environment and shifts to a new state. The critic holds the estimated value function and computes a TD error using the state of the environment and the reward obtained from the latest action. The error then updates both the estimated value function and the actor, which uses it to change the policy. A positive TD error results in enforcing the current action choice, while a negative one inhibits it. Figure 2.2 shows a schematic representation of the architecture.

While there exist off-policy approaches to the method, such as Degris et al. (2012), the base version uses an on-policy approach (Sutton and Barto, 2018).

Q-Learning

Q-Learning updates the action-value function of the current state using the greedy policy for the next step. The agent chooses an action a that transitions the environment from state s to

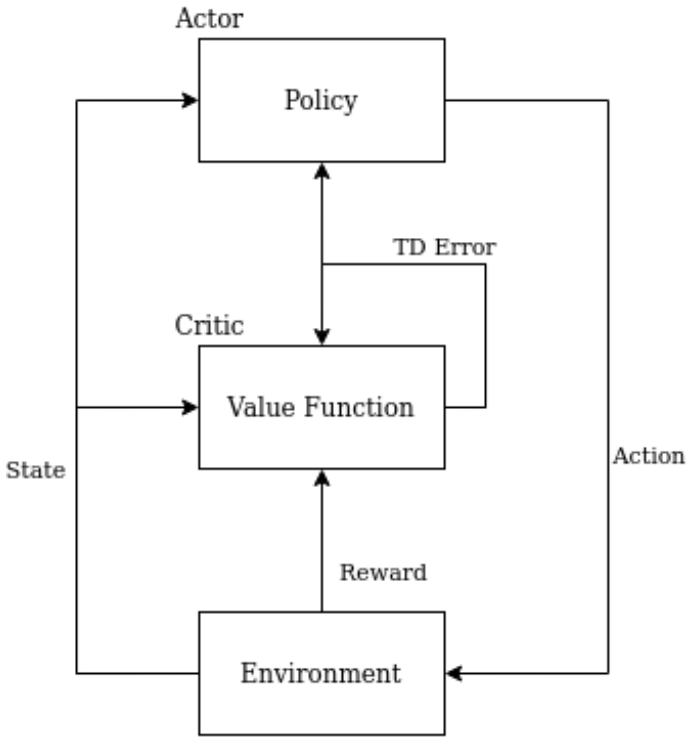


Figure 2.2: The Actor-Critic architecture

state s' using the policy under training, and then the action-value function gets updated using (2.15).

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2.15}$$

This update rule is similar to SARSA, with the difference that a' is chosen off-policy, in this case by the means of a greedy evaluation policy.

Proximal Policy Optimization

The Proximal Policy Optimization (PPO) method (Schulman et al., 2017) uses a concept called the trust region. At each step, the next update represents a new point present within a certain distance of the current estimate. The size of the trust region is adjusted accordingly, such that the policy does not change too much between each iteration. Adding soft constraints to the objective function, the second-order derivative matrix can be substituted with the first order, thus reducing the computational complexity. The learning is performed on policy.

Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) method combines the actor-critic approach with the deterministic policy gradient method (Lillicrap et al., 2019). The resulting algorithm is designed to be applied on a continuous action space. DDPG can also be considered an adaptation of Q-Learning to action or state spaces of high dimensionality, or even continuous ones. A replay buffer holds the state transitions gathered from experience, and the data is then used to train a neural network model. Due to the continuous nature of the action space, the gradient can be computed in a straight-forward manner.

A variant of the algorithm, called Twin Delayed DDPG (TD3), addresses the main issue of the basic implementation, namely the overestimation of the Q-function. To address it, TD3 estimates two Q-functions instead of one. Also, the policy gets updated less frequently than the

Q-function. Finally, noise is added to the target action, to prevent exploiting the existing errors in the Q-function. As a result, this method can only be used for continuous action spaces.

2.5 Hierarchical Reinforcement Learning

Classical RL methods contain different flaws that prevent it to be applied to more complex environments (Flet-Berliac, 2019). These drawbacks include problems with scaling up, generalization and abstraction. For instance, classical RL cannot be implemented for problems with a very large action or state space in an efficient way, whereas HRL can decompose it to smaller problems. If one agent's experience needs to be transferred to a different environment, this fails, as the agent is overspecialized to perform the task it was trained for. Finally, breaking down the main problem into multiple sub-tasks generally results in solutions that can be solved in a more simple manner.

In HRL, the policy is made up of multiple layers, each one acting at a different abstraction level. The lower layers perform elementary actions, for instance motor control, while higher layers are responsible for dictating sequences of lower level actions, or setting sub-goals for the sub-policies.

Next subsections present different basic approaches to HRL. While multiple approaches exist, of interest for this project are Feudal Learning and Options Framework. More advanced algorithms based on these approaches are explained later, in Chapter 3.

2.5.1 Feudal Learning

Feudal Learning is inspired by the medieval Europe's feudal system. Here, each layer represents a manager that assigns intermediary goals to sub-managers, who then do assign other subgoals to their sub-managers, until the bottom layer, that directly interacts with the action-space. Each managerial level observes the environment at different resolutions and achieves rewards upon reaching the goal imposed by its own manager. An interesting consequence is that the sub-managers only receive positive rewards if they do manage to fulfill the goal imposed to them, even if their actions result in accomplishing the manager's goal.

2.5.2 Options Framework

In the options framework architecture, the top level policy takes observations from the environment, and outputs one of the existing sub-policies, choosing one from the possible options. Each lower-level policy represents a sub-policy, which functions as a regular policy from the classical RL approach: it reads the observations from the environment and outputs actions from the agent's action-space. In other words, the action-space for the top level policy represents the possible options represented by the different existing sub-policies.

The architecture is summarized in Figure 2.3. Both the master policy and all the subpolicies have knowledge about the observation from the environment, and the master policy chooses one subpolicy to apply an action to the environment.

2.6 Deep Learning

Conventional ML methods do not perform well with raw data. Alternatively, custom designed features can be used as inputs to these algorithms, however this is not always an easy task. To learn complex high-level models, Neural Networks (NN) can be used. A NN is a multi-layered structure made of multiple nodes named neurons. Each neuron performs a weighted sum of its inputs and applies a nonlinear function on the result. The output of each such layer is passed forward to the next layer. During training, the weights of the neurons are adjusted in order to minimize the cost function.

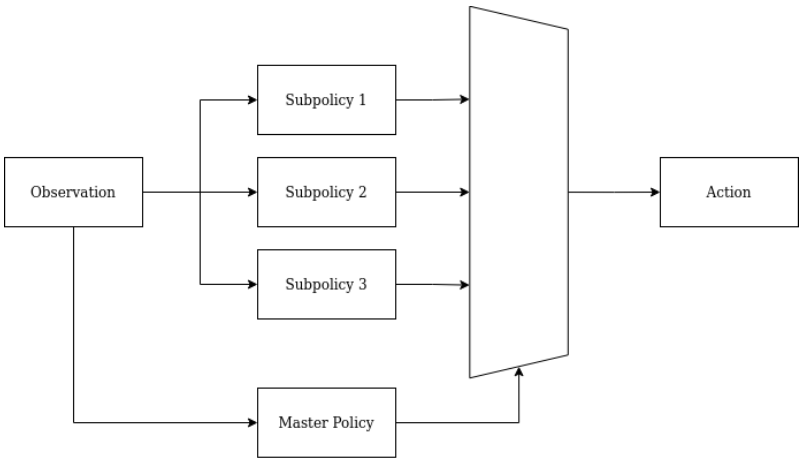


Figure 2.3: The options framework architecture. An action is chosen using a multiplexer controlled by the master policy

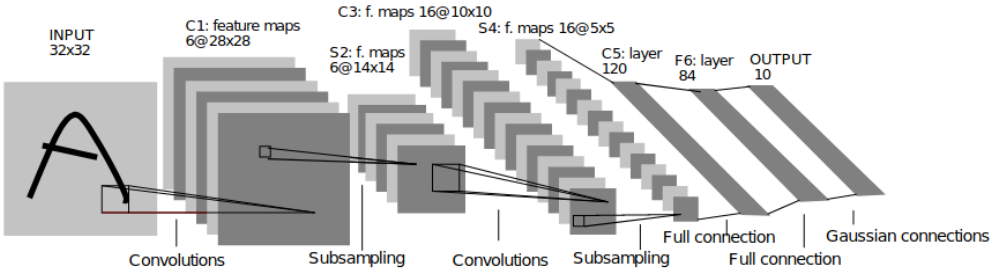


Figure 2.4: Sample CNN architecture. Image taken from LeCun et al. (1998)

In some cases, a simple NN is not enough to solve the RL problem. In the following subsections, the types of NN's relevant for this project are presented.

2.6.1 Convolutional Neural Networks

In the cases when the input of the network is an image, the number of inputs would be too high for a simple NN. Therefore, a different structure needs to be used. This is where Convolutional Neural Networks (CNN) are introduced.

The architecture of a CNN consists of convolution layers, pooling layers and fully connected layers. The convolution layers apply a convolution operation on the input image, transforming it into a structure called a feature map. Afterwards, the pooling layer samples from the feature map, reducing the number of dimensions of the feature map. The convolution and pooling operations can be repeated a few times, reducing the dimension of the feature matrix even further. Finally, after the last pooling operation, the resulting output is flattened into a one dimensional array and passed to the fully connected layers, which function as a standard NN with all its neurons from each layer connected to the ones on the next layer. An example of a CNN architecture is shown in Figure 2.4.

2.6.2 Recurrent Neural Networks

The Neural Networks presented so far use a simple feed-forward approach. This means that the output of one layer is only passed to the next layer. The drawback of this method is that, in the case of time series data, no information about the inputs from the previous time steps is available. This is where Recurrent Neural Networks (RNN) come in handy. These types of NNs contain feedback connections, thus using outputs from layers of previous evaluations of the

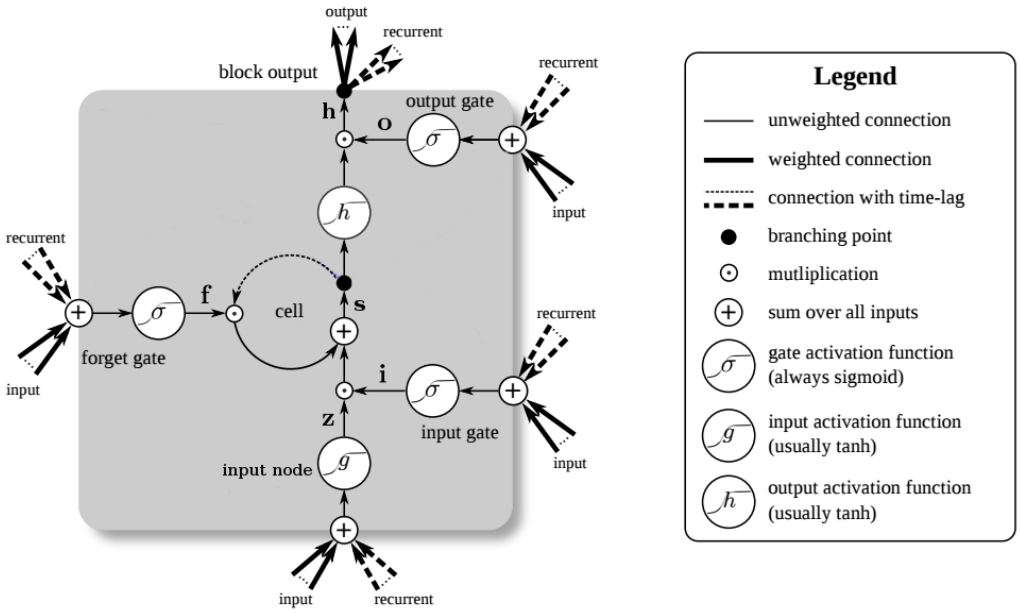


Figure 2.5: The structure of an LSTM unit. Image taken from Singh (2017)

networks in the current evaluation. This results in a state vector which acts as a memory of the previous elements from the sequence.

One example of a RNN is the Long Short-Term Memory (LSTM). In an LSTM, the neurons are replaced with LSTM units. This solves problems related to vanishing gradients. An LSTM unit is formed by a cell, containing the main feedback structure, and three gates responsible for regulating the information flow to and from the cell - an input gate, an output gate and a forget gate. Figure 2.5 shows the structure of an LSTM unit.

2.7 PIRATE Robot Overview

The PIRATE robot is an autonomous pipe inspection robot in the shape of a snake whose purpose is to explore pipe networks of different diameters and to check for defects in the pipe's inner walls. The electromechanical construction of the robot is represented by seven sections separated by rotating first order joints. Each joint is equipped with a Direct Current (DC) motor able to rotate the joints. Their purpose is to allow the robot to clamp inside the pipes. Furthermore, each one of the joints has a wheel attached to it, that is controlled by a separate motor and allows the robot to move forward and backward. The middle module is split in half by another rotating joint, which allows the robot to rotate around its longitudinal axis, and thus helps in orienting the whole robot inside the pipe. The front module is equipped with a camera. The data recorded by the camera is used for navigation purposes, besides the normal function of detecting pipe defects. A schematic representation of the robot is shown in Figure 1.1.

For this project, the physical robot is not used. Instead, all the experiments are run inside a simulator, where the dynamic behaviour of the robot is simulated.

2.8 PIRATE Previous Work

At the start of this project, the PIRATE robot can perform autonomous navigation through straight pipe segments of constant diameter and 90 degrees turns. This section describes the state of the navigation controller.

The navigation controller adopts a HRL structure using the options framework. One master policy chooses between three subpolicies, each responsible for one type of movement: one for

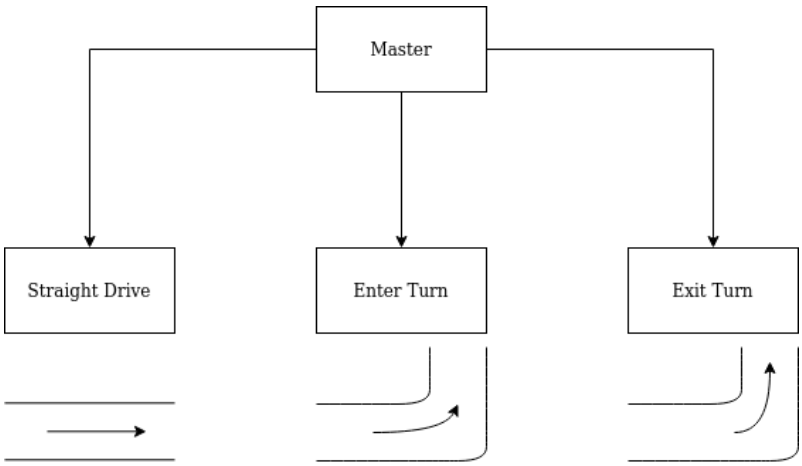


Figure 2.6: Initial HRL structure with 3 subpolicies

driving through straight pipes, one that allows entering turns, and another one for leaving the pipe bends. Figure 2.6 shows a schematic of this initial HRL structure.

Each policy uses a Convolutional Neural Network (CNN) model at its base. The policies use the camera information, which represents a 64 by 64 image on 3 channels (red, green, blue). The data coming from the camera is a color coded depth map - yellow showing that the pipe wall is closer, and blue that it is further away to the camera sensor situated at the front of the robot. The training is performed using a PPO approach.

Besides the camera, the policies use as part of the observation space the positions and orientations of the robot joints, as well as the previous action taken taken by the policy. Some policies may also contain other type of information in the observation. The type of data present in the observation space is discussed in the corresponding subsection.

In terms of action space, the top level policy picks between one of the possible subpolicies, in an options framework approach. In comparison, the subpolicies output actions which are passed to the robot controller. They are used to dictate a setpoint value to either the motor velocity or joint position of the robot. In general, the controlled values represent the velocities of the outer wheels of the PIRATE, or the angular position of the joints. One exception is that one of the angular joints that assist in clamping the robot receives a constant velocity in all the subpolicies, to ensure that the robot stays clamped. This is important especially in vertical pipe segments.

2.9 CoppeliaSim Simulator

CoppeliaSim is a robot simulator that can be used for simulating dynamic behaviour. It is based on a distributed control architecture, thus allowing controlling the simulations using remote APIs or other types of scripts. It provides out of the box support for a remote API in multiple languages, including C, Python and Matlab, and a regular API for C. These allow control of the simulation from outside the simulator, for example pausing, starting, stopping, control of different joint positions and velocities, and so on. Via the remote API, a real robot (or another PC) can control the simulation running on a main server.

In order to simplify the development process, and at the same time benefit from the speed of the regular API, a Python wrapper is used to control the simulation environment. The capabilities of the API include adding new components in the simulation using stereolithography (STL) CAD files. This is useful for generating a random pipe network, thus allowing the RL algorithms to learn from a multitude of configurations.

2.10 Summary

In this chapter basic background information about the tools used in this project is presented. Firstly, the concept of RL is introduced and compared with other types of ML in Section 2.1. Then the topic of RL is expanded in Section 2.2, starting with the basic MDP, continuing with the definition of the value function, the Bellman equation. A method that shows how to solve the MDP by learning a deterministic policy using Dynamic Programming is described in Section 2.3. Afterwards, sampling methods that allow training a policy without knowledge of the structure of the MDP are introduced in Section 2.4. The topic of RL is closed with the presentation of the HRL approach and deep learning methods (Sections 2.5 and 2.6). Finally, the last sections show an overview of the current implementation of the PIRATE robot, alongside the work on which this project is based, and the CoppeliaSim simulator used to run the experiments.

3 Related Work

In this chapter, different state of the art algorithms and methods for implementing a HRL controller are presented. These methods are built upon simpler algorithms, described in Chapter 2, and offer insights about different architectures and approaches for building a HRL structure. Furthermore, other research related to pipe navigation robots, including previous work on the PIRATE, are shown in this chapter.

3.1 Feudal Networks

Feudal Networks (Vezhnevets et al., 2017) represent an architecture based on Feudal Learning (see also Section 2.5.1). The architecture consists of a modular neural network, with two independent modules. Similarly to the classical Feudal Learning approach, there is a manager that decides on intermediate goals, formulated as directions, while the worker interacts directly with the environment by taking actions, thus implementing simple behavioural primitives. As the policy of the worker should ultimately lead to reaching the goal directions set by the manager, the authors decided it is efficient to use a form of policy gradient method to train the higher level agent, while the lower agent uses an intrinsic reward function.

3.2 Hierarchical Reinforcement Learning with Off-Policy Correction

The HIerarchical Reinforcement learning with Off-Policy correction (HIRO) method (Nachum et al., 2018) uses a two-layer structure of HRL. As in other approaches, the top level agent sets goals for the lower level agent. The issue the authors identified with other methods is the fact that on-policy training of the lower-level policy creates a non-stationary problem for training the higher-level policy. In other words, transitions obtained from past experiences are not representative for the future shape of the policies. Therefore, an off-policy correction is added to increase the accuracy of the past experiences with respect to the current policy. To achieve this, different goals are evaluated in terms of the log probability to achieve state transitions more suitable to the current policy. The goal with the highest log probability is then used to relabel the current transition.

The schematic representation of the method is presented in Figure 3.1.

3.3 Hierarchical Actor-Critic

One issue in training HRL models is the fact that training policies from different levels of the hierarchy in parallel is a slow process. As the lower level policies change, the higher level policies become harder to train, as the transition functions are not stable. After a time, the policies converge to a value that no longer fluctuates, and this makes learning multiple policies feasible. The purpose of the Hierarchical Actor-Critic (HAC) architecture (Levy et al., 2017) is to speed

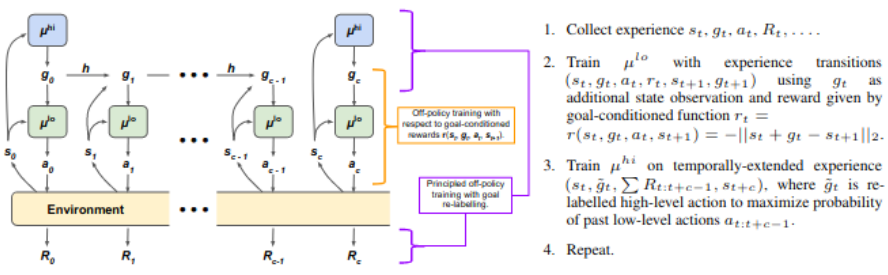


Figure 3.1: The design of HIRO. Image from Nachum et al. (2018)

up this training using a hierarchical structure, where each hierarchical level outputs subgoals used on the next level, and a method that allows learning using sparse rewards.

In order to train multiple policies using sparse rewards, a failed training step is relabeled such that the subgoal becomes what the agent managed to achieve for that iteration. The result is that each lower-level policy is considered to be already optimal when training higher level policies.

3.4 Hierarchical Deep Q Network

The Hierarchical Deep Q Network (h-DQN) method (Kulkarni et al., 2016) adapts the DQN approach to a hierarchical model. The DQN itself is an implementation of a Q-Learning algorithm to a deep learning architecture. A neural network is responsible for mapping states of the environment to actions in a non-linear way. Finally, the hierarchical structure implies the existence of a higher-level policy trained over subgoals, and a lower level policy that implements the goals and takes primitive actions on the environment.

3.5 Pipe Navigation Robots

The PIRATE robot is presented in Dertien (2014). The motivation behind the presented design, namely the snake-like shape of the robot, is building a modular pipe inspection robot that can move through pipes of various diameters. Its modular construction and snake-like shape allow the robot to take turns in the pipes, and navigate through T-joints or inclined pipes. The research also shows that a human operator can operate the robot remotely by using simple moves. These commands, which represent, for example, clamping the front of the robot, rotating, or driving the back wheel, have to be combined to achieve more complex movements: having the robot drive forward, rotate inside the pipe, or take a turn.

Later research focuses on making the PIRATE robot navigate autonomously. For example, Garza Morales (2016) proposes a software architecture using the Robot Operating System (ROS) framework that allows controlling the robot in an autonomous way. In later work, Grefte (2020) uses a special type of RL to train different policies for each type of complex operations: entering a turn, leaving a turn, and driving forward. Another main policy is responsible for choosing between the three sub-policies.

Besides the PIRATE robot, there are a multitude of pipe inspection robots designed by different groups. While some of these robots, like the PIRATE robot, employ a snake-like shape for the mechanical design, and use clamping to navigate the pipes, other approaches exist as well. For instance, Nassiraei et al. (2007) introduce the KANTARO robot. It drives through the pipes using wheels whose configuration adapt to the shape of the pipe. Therefore, the robot can drive through Y-junctions and bends, and it employs relatively simple mechanical structure. However, due to its construction, the robot cannot enter the side pipe of a T-junction or climb vertical pipes. An image of the robot can be seen in Figure 3.2.

3.6 Reinforcement Learning in Robotics and Parameter Tuning

Reinforcement Learning (RL) is used for solving navigation issues for different types of robots. Kahn et al. (2018) present a method that allows robots to learn to navigate a complex environment using a self-supervised approach. Using a reward function, policies are trained via a combination of model-based and model-free learning methods. Data acquisition is performed using a camera mounted on the robot.

Each ML model contains different hyperparameters that can affect the quality of the final trained model. One way to tune these models include performing parameter sweeping (Wang et al., 2019). In this approach, the search space of the parameters is divided into a grid and different models are built by sweeping through the parameter grid. The model that has the best

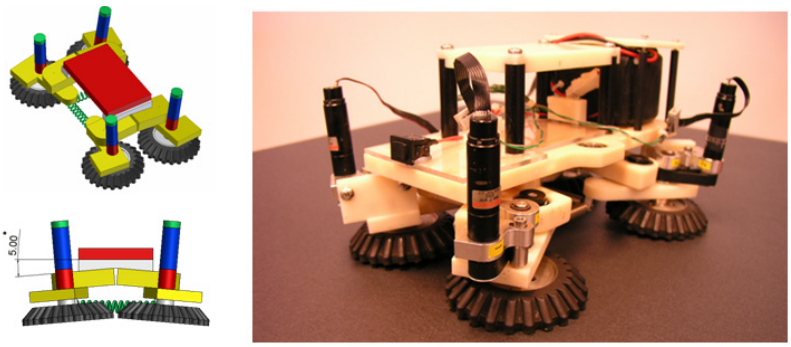


Figure 3.2: The KANTARO pipe inspection robot. Image taken from (Nassiraei et al., 2007)

performance is kept. Another approach is Population Based Training (PBT) (Jaderberg et al., 2017). The idea is to train multiple models in parallel, and periodically update the parameters of the ones performing worse with the hyperparameters of the better performing ones. The performance of the trained models is considered when evaluating a specific parameter combination using a specified metric.

3.7 Summary

Multiple methods of training HRL models exist, which employ different architectures. The methods presented in this chapter focus on training a hierarchical structure for which the high level agent dictates subgoals for the low level policies. The methods themselves are based on principles presented in Chapter 2. Because the methods themselves are not designed to train an HRL architecture based on the options framework, they are not used in this project.

This chapter also presents related work in the field of autonomous pipe inspection robots, as well as other methods to apply RL in robotics besides a hierarchical approach. From this information, the results of the previous work on the PIRATE robot are important for this project, as they represent the starting point of this research. Finally, parameter tuning using PBT is another subject presented in this chapter which is useful for the work presented in the next chapters.

4 Methodology

In this chapter the different types of experiments performed in the project are presented. They focus on searching for an effective method to extend the functionality of the navigation control of the PIRATE to be able to steer inside T-junctions. Furthermore, the applications of this new feature are also explored, namely traversing a previously mapped pipe network and exploring an unknown one.

4.1 Hierarchical Structure Adaptation

One drawback of the architecture of the navigation controller presented in past work is that it does not take into account possible T-junctions in the pipe network, which requires a different decision strategy. To compensate for this, various methods are explored in this project. Common to all the approaches is adapting the master policy to provide a subgoal to the subpolicies. During training, the top level agent receives a reward for reaching the waypoints inside the pipe network, dictated by path planning (see also Section 5.2). In case of a T-junction, the top level policy needs to learn to decide between the two possible turns and to choose the appropriate subgoal that achieves this task. Therefore, the HRL structure of the navigation controller is adapted to a combination between the options framework and the feudal learning approaches.

The first approach makes use of a new subpolicy, responsible for orienting the robot properly inside the pipe. The subgoal for this policy represents the orientation angle inside the pipe for the robot. In the second approach, the enter turn subpolicy is adapted to support a subgoal, which represents a binary value with one value if the robot should steer to the left, and a different one if the robot should steer to the right. In this context, left and right are defined relative to the orientation of the straight segment before the bifurcation. In the case when the T-junction is placed longitudinally, with its long side continuing the previous straight segment, left and right depend on the orientation of the bend. Section 4.3 describes the possible subgoals for the turn policy in more detail.

Figure 4.1 shows a schematic of the updated HRL architecture with 4 subpolicies and one subgoal, while Figure 4.2 shows the structure using only 3 subpolicies and one subgoal.

4.2 Rotational Subpolicy

The purpose of the rotational subpolicy is to aid the PIRATE robot rotate inside the pipe. The policy has a 6 dimensional action space, one for each rotational joint of the robot (see also Section 2.7 for the mechanical construction of the PIRATE). Each joint receives a setpoint angle between $-\pi$ and $+\pi$. During experiments, one of the actions is replaced by an angular velocity provided to the second joint, in order to force the robot to clamp while rotating. In terms of

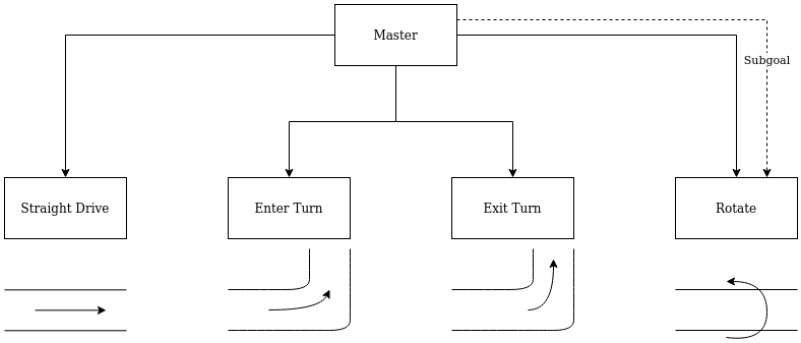


Figure 4.1: Updated HRL structure with 4 subpolicies and one subgoal

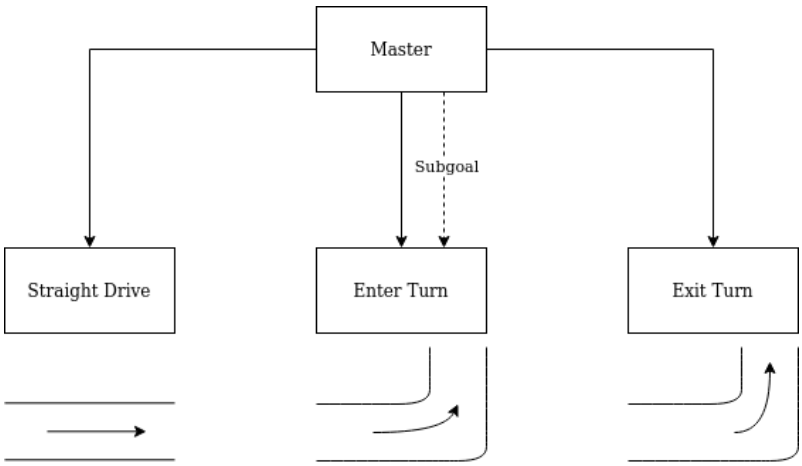


Figure 4.2: Updated HRL structure with 3 subpolicies and one subgoal

the observation space, the camera information is coupled with the positions and orientations of the wheels and joints, along with the relative orientation of the robot inside the pipe and the actions taken at the previous step. The reward received by the policy is a combination between the angle difference between the rotational goal and the target goal, and a clamping reward.

From manual tests, the robot can rotate efficiently by firstly clamping the front (or back), rotating the middle joint, then unclamping while clamping the other side, and finally reverting the middle joint to a zero rotational angle. To encourage the robot to behave in a similar manner, the reward has a clamping component, which represents the distance between two wheels (the first and the third, starting from either end). The lower the value is, the more clamped is the robot. The clamping component considered for the rotational experiments depend on the back clamping distance. Section 6.5 shows the result of this approach. The formula for the reward at timestep t R_t is shown in (4.1). Here, p_t^i represents the position in space of the i-th wheel at time t, with 0 being the wheel at the back of the robot and 5 corresponding to the front-most wheel; θ_t is the relative rotation on the Z axis of the front of the robot with respect to the pipe; and θ_g is the rotation goal. The choice of the value of the exponent for the clamping component is attributed to previous work.

$$R_t = -\|p_t^0 - p_t^2\|^4 - 4|\theta_t - \theta_g| \tag{4.1}$$

The top level policy can generate a subgoal in a Feudal Learning approach, thus dictating a desired orientation of the robot inside the pipe. This should allow the robot to prepare for taking a turn in a normal corner or a T-junction. The subgoal is considered to be reached if the angular difference between the robot and the target rotation is less than 5 degrees.

In case of the individual experiments, when the policy is trained individually, the rotational target is the orientation of the first corner segment (or T-junction segment) with respect to the pipe where the robot is situated. In a HRL structure, the rotational goal is generated through Feudal Learning by the top level policy.

4.3 Enter Turn Subpolicy

The second approach for T-junctions decision making is to adapt the enter turn agent to accept a subgoal. During training, one episode is considered to be successful if the robot manages to reach the correct target. In case of a simple bend, the target represents the straight pipe element just after the bend. For a T-junction, the environment picks a random target at the start of the episode between the two possible options. The corresponding target position that the robot needs to reach is then saved into the environment state, and used when evaluating the possible

end of the episode. If the PIRATE gets close to this position, the agent receives a positive reward and the number of successful episodes is incremented. In the other case when the robot takes the wrong turn, a penalty term is subtracted from the reward of the current episode. Besides the episode end reward (or penalty), the reward also has the same clamping reward used for other policies.

4.3.1 Reward

Equation (4.2) shows the formula for calculating the reward R_t at the timestep t. In the equation, D_t is the distance between the front of the robot and the target position; σ_R, σ_G and σ_B are the standard deviation for each channel (red, green, and blue respectively) of the pixels from the image observation; finally, R_e is the end episode reward, which has a value of 5000 if the episode ended successfully, with the correct target reached, -5000 if the wrong target was reached, or 0 if the episode did not end yet. Furthermore, the elapsed time for each episode is tracked, and a reset is called if the agent does not reach the target in a certain amount of time, thus starting a new episode and preventing the agent from remaining stuck.

$$R_t = 1000(D_{t-1} - D_t) + 10(\sigma_R + \sigma_G + \sigma_B) + R_e \tag{4.2}$$

4.3.2 Action Space

The action space consists of one velocity applied to the last joint, to force the robot to clamp, five orientations for the rest of the joints, and three velocities applied to the three back-most wheels, with the rest of the wheels blocked at a zero velocity. Some experiments are performed with an extended action space, outputting wheel velocities for all the six wheels of the PIRATE.

Another variant of the action space considers the fact that the robot might get stuck. In the normal implementation, the PIRATE is forced to go forward, as the motor controller passes the absolute value of the velocity action. Instead, if more than half of the wheel velocity actions are positive, the motor controller passes a positive velocity to all the controlled wheels, driving the robot forward, otherwise the setpoint velocity has a negative value. This is to prevent the robot from experiencing longitudinal forces due to the wheels turning in opposite directions. Instead of using this trick, some experiments employ an additional action, whose sign dictate whether the PIRATE should drive forward or backward.

4.3.3 Subgoal

The following subsections show different ways of implementing the subgoal. Regardless of the shape of the subgoal, a corresponding value is also added to the observation space of the agent.

Binary Subgoal

In this approach, the goal has a binary value, zero representing a left turn and one a right turn. Left and right are considered relative to the orientation in space of the pipe bend, left representing a relative orientation of $-\pi/2$ of the target pipe with respect to the straight pipe before the turn, considered on either X or Y axis; while right is a relative orientation of $+\pi/2$ of the same pipes. For consistency purposes, the convention is applied to both T-junctions and pipe corner elements.

As the agent needs the information about in which direction to steer, the current subgoal is added to the observation, translated to a value of either $-\pi/2$ for a left turn, or $+\pi/2$ for a right turn. Alternatively, left and right can be encoded also as a value of 1, respectively 2. In general, the experiments do not use this variant, unless mentioned.

Subgoal Value	Orientation Vector
0	$[0, \pi/2, 0]$
1	$[0, -\pi/2, 0]$
2	$[\pi/2, 0, 0]$
3	$[-\pi/2, 0, 0]$
4	$[0, 0, 0]$
5	$[\pi, 0, 0]$

Table 4.1: The possible values of the directional subgoal, along with its orientation vector

Directional Subgoal

Instead of using a simple flag for the subgoal, this approach considers 6 possible values of the subgoal, one for each possible orientation of the straight pipe elements. From the construction of the pipe network, the pipe elements can only be oriented on the positive or negative half-axes of the X, Y and Z axes of the space.

The subgoal is encoded into an orientation vector in space, represented by Euler angles. This is due to the fact that the orientation of the robot is also expressed in terms of Euler angles. This vector is added to the observation space using the values shown in Table 4.1. The third value of the vector remains zero, as it represents the roll, which is not relevant for this experiment.

Position Subgoal

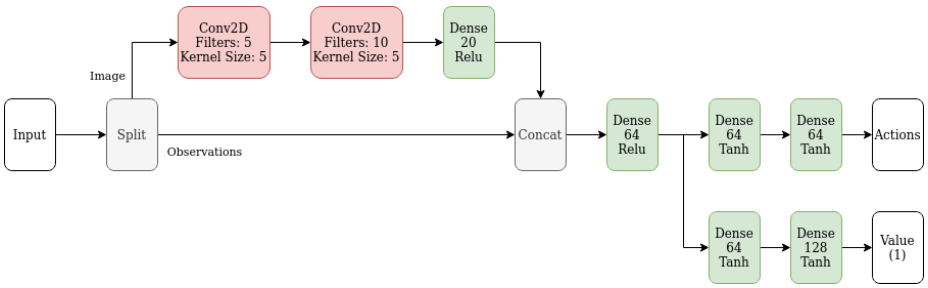
It is also possible to pass the actual position of the target to the agent as part of the observation. This approach offers more information to the RL agent. However, it is way more complicated for the top-level agent to calculate the subgoal in this way. One idea is to estimate the position of the theoretical pipe elements using a directional subgoal approach. This means that the top-level agent still generates a subgoal having 6 possible values, with the difference that the corresponding vector is a position calculated from the current position of the PIRATE. While this does not offer an exact target position of the subgoal, the correct subgoal would be closer to the real target than the rest of the values.

Because of the complexity of calculating the right position starting from a subgoal and the reduction in generalization, this approach is only considered for comparison purposes in the later experiments.

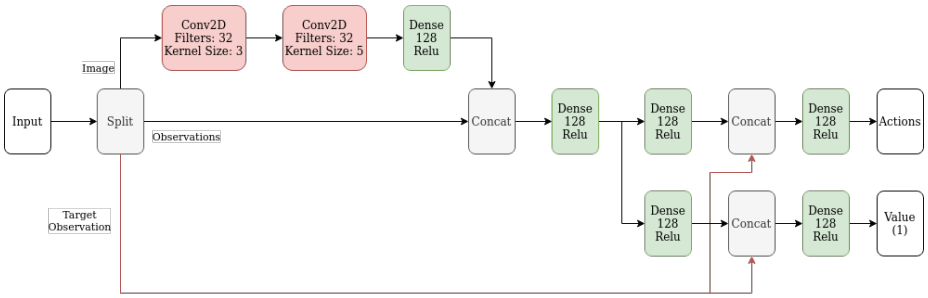
4.3.4 Curriculum Learning

Steering into a T-junction starting from a random orientation of the robot with respect to the junction itself is a complex task. This is also proven by the results from Section 6.6. The idea of curriculum learning is to start from a very simple case and increasing the complexity gradually. Therefore, the problem is simplified from the initial generalized case, and specific, simpler cases are tried to be solved first. The pipe system environment is on a horizontal configuration, and the robot is placed closer to the goal, directly inside the T-junction.

After these initial considerations, the first step is to place the robot in the pipe in such a way that it is oriented directly towards the goal. After training an agent able to succeed in this trivial scenario, the same agent is retrained on different orientations of the robot. The second step of the curriculum learning approach is retraining the previous model on an environment where the robot is spawned oriented between the two sides of the T-junction. In this way, the agent decides whether the robot should orient itself towards the left or the right. Therefore, it can be tested whether the agent can use the target information properly.



(a) The old model used in Grefte (2020)



(b) The new model, including the target observation

Figure 4.3: The old and new CNN models used for the turn agent policy

The third step is to move the target position further away from the center of the T-junction, in order to advance further. If the turn agent manages to take the right decision (as part of step 2) and bring the front half of the robot past the junction element, then it accomplishes its goal.

4.3.5 Policy Model

The CNN model employed for the policy uses two convolutional layers of 5, respectively 10 filters, both using a kernel size of 5. This results in a relatively small number of features extracted from the camera. Because of the newly added T-junction, this might not be enough for the agent to get the required information from the environment in order to reach the correct target. Therefore, the number of filters for both layers gets increased to 35, while the kernel size of the first convolutional layer is decreased to 3. In this way, the resulting feature map will have a higher dimension.

Because of the increase in the size of the feature map, it is also important to increase the size of the hidden layers as well. The previous model employs 64 neurons on each of its hidden layers, and uses as an activation function the hyperbolic tangent (tanh). The size of the layers gets increased in the current model to 128, and the activation function becomes the rectified linear unit function (relu). These modifications ensure that the there are enough neurons in the network to properly process the newly increased feature map.

The old CNN model is presented in Figure 4.3a, while the modified one is shown in Figure 4.3b. The observation related to the target is extracted from the rest of the observations and is passed into the last hidden layer of the network. In this way, it should have a higher impact on the output of the network.

All the agents described so far use a CNN implementation for their policy. In case of the turn agent, a recurrent model is also considered during some of the experiments. Before passing the observation matrix in the model, it gets flattened, obtaining a one dimensional vector.

4.4 Pipe Traversal

One use case for the PIRATE is to navigate through an already known pipe system. Considering a previously explored network, a map can be built, stored in the memory of the robot controller. This can be used to aid the robot to navigate on a predefined path, while trying to reach a desired end position inside the pipe system.

The map of the pipe network is generated using the algorithm described in Section 5.1, and the path towards an end goal is constructed using the method presented in Section 5.2. The result is an array of waypoints that the robot needs to reach. The navigation controller then generates actions that allow the robot to drive and cross through bends and T-junctions inside the pipe network.

The top level policy is trained for achieving this task. As each waypoint from the calculated path is reached, the agent receives a small reward, with a bigger one awarded for arriving at the end position. On the other hand, in the case where the robot gets too far from the next sub target, a penalty is applied to the current reward, and a new episode is launched. The same penalty is applied in the cases where the robot gets stuck, for example in T-junctions where the PIRATE was not able to steer properly. This ensures that the robot remains straight when navigating, preventing bends beyond the capacity of the mechanical construction - which are possible because of the simulation.

4.5 Pipe Exploration

Section 4.4 describes the case where the pipe network is already known beforehand. However, in real situations this is not often the case. Older pipe systems might not have a digital map which shows the pipe network in its entirety. In these kinds of scenarios, it is important for the PIRATE to be able to navigate inside an unknown environment, and map the pipe system from the inside for future inspection missions. Building a 3D model or performing a full mapping of the pipes is beyond the purpose of this project, therefore the focus falls on training a navigation controller model able to explore as much of the pipe network as possible.

This task is achieved by adapting the top level policy to receive a reward for exploring more pipe elements present in the network. Starting from a closed pipe network (see also Section 5.1.3), the robot navigates through the pipes. As new elements are reached, the RL agent is awarded a reward based on the update of the exploration percentage. The way this metric is calculated, as well as how it is updated, is described in Section 5.1.4. In this case, as there is no end target position, no reward is received for reaching a specific pipe element.

4.6 Summary

In this chapter the various experiments performed to answer the research questions proposed in Section 1.3 are described. Section 4.1 details the modifications of the previous HRL structure to adapt to generating a subgoal, thus transforming the architecture from the simple options framework to a combination of the options framework and feudal learning. The options are to add a new policy, responsible for rotating the robot in the right position (see Section 4.2), or to adapt the existing turn policy to accept a subgoal, representing the steering direction (see Section 4.3). The former approach is further split into using different types of subgoals, action spaces, and policy models.

Adding T-junctions to the pipe network opens the possibility to investigate practical uses of the PIRATE, namely traversing an already known pipe network while trying to reach a certain position (treated in Section 4.4); or exploring a completely unknown pipe system (see Section 4.5).

5 Experimental Design

In this chapter, different aspects of setting up the experiments are presented. Section 5.1 shows the way the pipe system is built in a procedurally generated way. Afterwards, Section 5.2 explains how a path is built through the already constructed network, which the robot has to follow during experiments. Section 5.3 describes the different configurations of pipe networks needed for training each particular RL agent. Finally, Section 5.4 presents the necessity of tuning the RL agents, and how this is performed.

5.1 Pipe Network

In the simulation, the PIRATE robot navigates through a system of pipes. For each episode of the training, the pipe network is randomly generated, in order to ensure a high variety of configurations. This contributes to the generalization and robustness of the models under training.

5.1.1 Building the Network

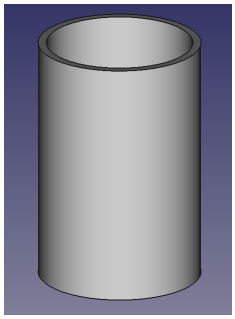
The pipe network is procedurally generated for each training episode. The building blocks of the network include a straight pipe segment, 90 degrees corners, a reducer and a T-junction (also called a tee). The reducer has an inner diameter increase of 20%, and the T-junction segment has all of its connections of the same diameter. All of the primitive pipe shapes are modelled in CAD, and are saved in the stereolithography (STL) format. The pipe elements used in previous work are presented in Figure 5.1a and Figure 5.1b, showing the straight pipe element and the corner pipe element. The new elements added in this project are T-junctions and reducers. Figure 5.1c shows the 3D model of the T-junction element, while Figure 5.1d the CAD design of the reducer pipe element.

When a new pipe element is added to the network, a mesh is loaded from the corresponding STL file using the CoppeliaSim API. This adds it in the simulation scene. It is important to set the correct position for each element, such that the robot will be able to navigate through a proper pipe network. To achieve this, the position and orientation of each element is set appropriately. For instance, in a straight pipe segment, each new straight element is placed with the same orientation as the previous one, and its position is the position of the previous element, shifted on its Z axis (its longitudinal axis) by an amount equal to the length of the element.

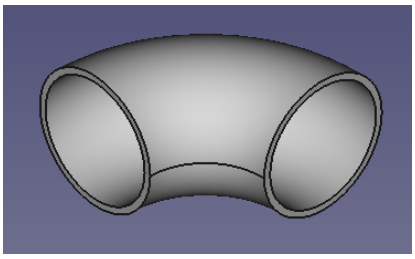
Each pipe network starts with a starting pipe, that sets the orientation of the first straight segment - either horizontal or vertical, pointing upwards or downwards. The first straight segment has a configurable length, and ends with either a corner pipe or a T-junction segment. A configurable parameter dictates whether the network contains sharp corners or not. Further straight segments have a random length, bounded by two configurable parameters. These lengths are measured in the number of straight pipe elements present on the segment.

A straight segment has a chance to contain a reducer pipe element. This either increases the inner diameter by 20% of its current diameter, or it decreases it to its normal value. To further randomize the pipe, with a certain probability the initial straight segment can be wider than the default value. Both of these probabilities are configurable. In the case a straight segment should contain a reducer, it is placed as close as possible to the middle of the segment. For example, in case of a length 4 segment, it is placed after the first 2, and in a length 5 segment it is placed after the first 3.

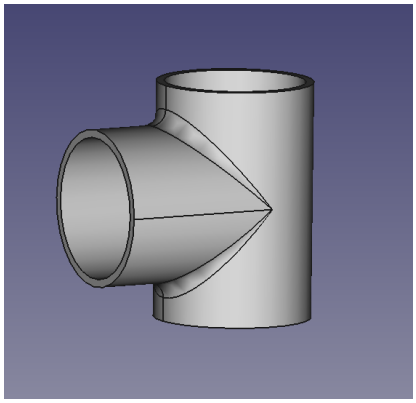
Each corner is placed such that it fits with the last element of the previous straight segment, and it gets a random orientation, multiple of 90 degrees. Therefore, the next straight segment can be oriented on one of four possible ways. Regarding T-junctions, there are two possible con-



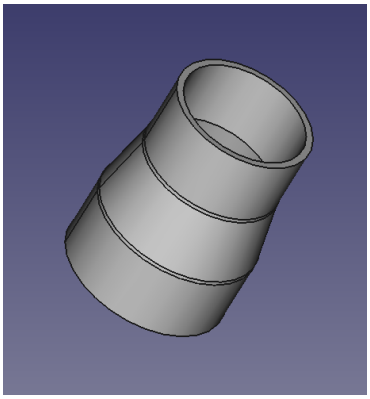
(a) A straight pipe element



(b) A corner pipe element



(c) A T-junction pipe element



(d) A pipe diameter reduction element

Figure 5.1: CAD designs of the used pipe elements

figurations, ignoring its rotation around the original direction: either the element is oriented longitudinally, with its long side continuing the straight segment, or transversally, with the two pipe openings oriented on a line perpendicular to the original orientation. In both cases, the pipe is further rotated similarly to the corner pipe, in one of the four possible direction. In the transversal case, due to the symmetry of the element, this only results in two possible configurations. To sum it up, the T-junction can be placed in a total of 6 different configurations. The probability of placing a T-junction instead of a corner, as well as placing the pipe element in a transversal instead of longitudinal orientation are configurable parameters of the generator.

After placing a corner or a T-junction, the generation continues with another straight segment starting from a random open end of the pipe, until the maximum number of segments is reached. This is another configurable parameter passed to the pipe generation module.

The pipe network is represented as an unoriented graph. Each node contains the position, orientation, name, index, and neighbors of a straight pipe segment. In case of a T-junction, the straight element just before the bend has two neighbors, the two straight elements which represent two new openings in the pipe network. The index of a node is the order in which the corresponding element is added to the pipe element name during generation.

This structure is useful for the target reaching experiments, as a path planning is required to generate waypoints that the robot needs to follow. This feature is discussed in more detail in Section 5.2. Another usage of the graph is the detection of collisions when adding a new element. A breadth-first traversal is used to compare the position of each element in the pipe with the new element that is added. In case of a collision, the whole network is destroyed and the generator starts from zero.

A sample pipe network generated using the presented method is shown in Figure 5.2. A diagram of a graph can be seen in Figure 5.3. In this image, each node, represented by a black dot, contains an index and has as neighbors the nodes which are connected to it. The element

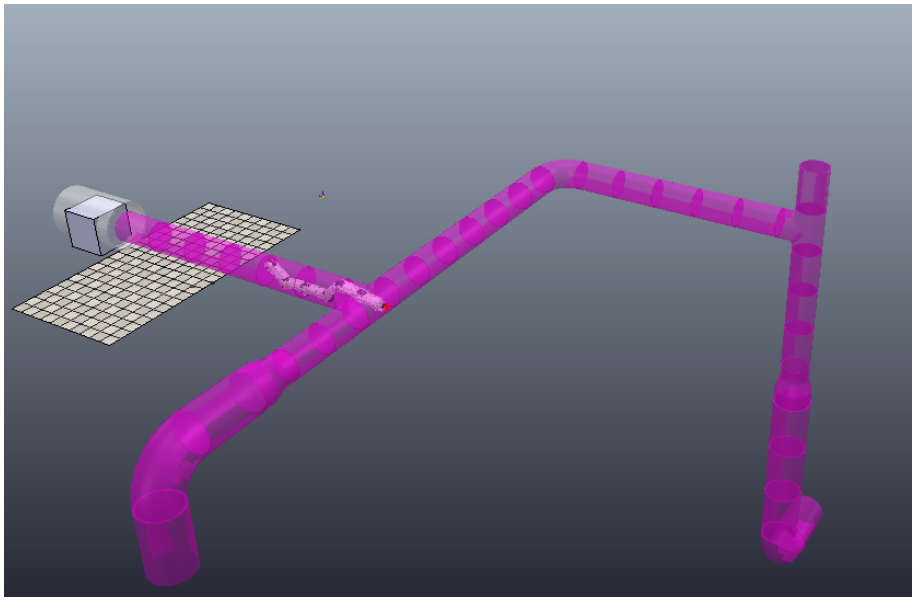


Figure 5.2: Sample generated open pipe network of length 5

corresponding to element 2 continues with a transversal T-junction, leaving 3 and 4 as openings in the network. For the next segment, 3 is chosen as a starting point, and the segment ends with a simple corner. Afterwards, 4 is chosen, and finally 7, which ends in a longitudinal T-junction. Note that while 2 has 3 and 4 as neighbors in the junction, 3 does not have 4 as a neighbor.

5.1.2 Static Pipe Networks

Some experiments require a pipe network that does not vary between episodes. By passing a special configuration, the orientation of the initial straight pipe segment, as well as the rotations of all the pipe bends placed during the pipe generation can be controlled. The information is passed by specifying, in order, a list of numbers between 0 and 3 dictating the relative orientation of each bend with respect to the previous straight segment. The exception is the first number, which controls the orientation of the initial pipe segment with respect to the X axis. The number itself is the multiple of $pi/2$ desired for the respective element.

For simplicity, it is considered that all straight segments have the same length, and all bends are T-junctions. This aspect does not reduce the generality, as mainly the static pipe generation in the project is used specifically for navigating T-junctions.

The static configurations used in this project depend on different ways to orient a T-junction with respect to the pipe segment from which the robot drives. If the robot enters the T-junction from the side hole of the junction, the pipe element is considered to be in a transversal configuration. Otherwise, it is a longitudinal one.

The main transversal static configuration used in this project is the horizontal one, where both the first straight segment in the network, as well as the sides of the T-junction are situated in the horizontal plane. Other configurations are the horizontal-vertical one, where the T-junction is placed perpendicular to the horizontal plane; the upwards-oriented, where the initial straight segment is oriented upwards; and, finally, the downwards-oriented, where the robot enters the T-junction from the top.

Figure 5.4 illustrates the possible static pipe configurations. In a transversal configuration, the robot enters the T-junction from the side arm of the pipe element. In a longitudinal configuration, the entrance point is either of the other two open ends.

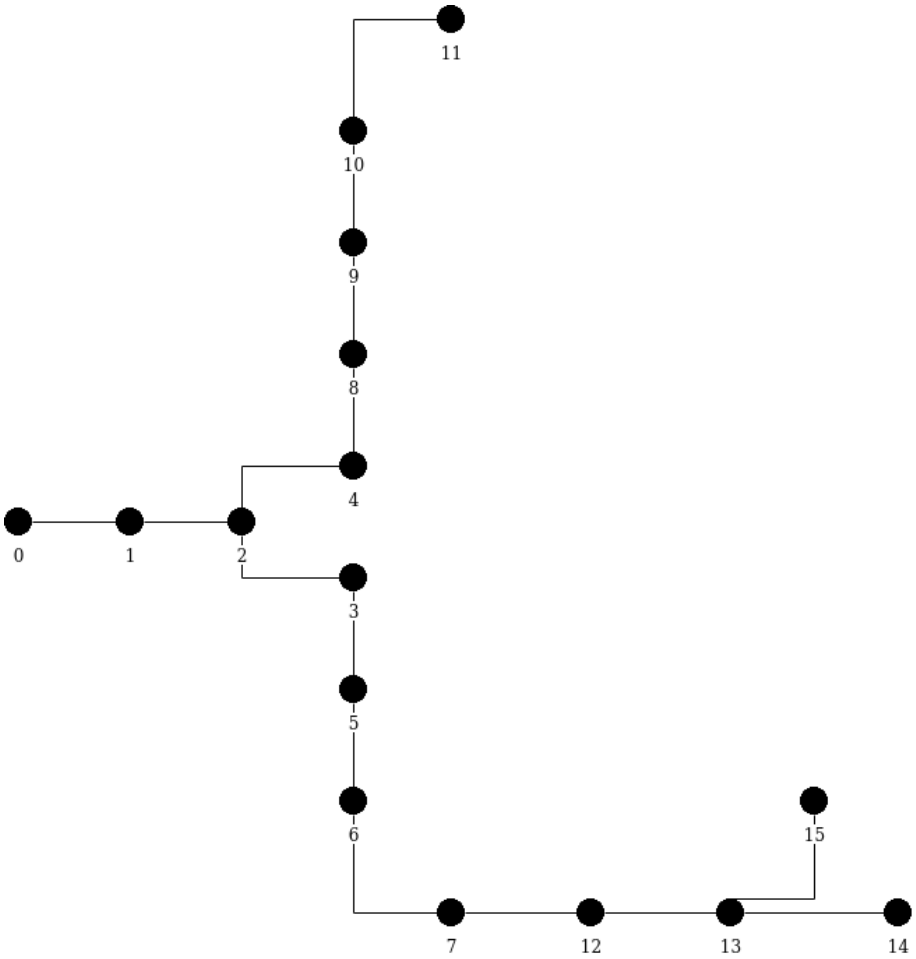
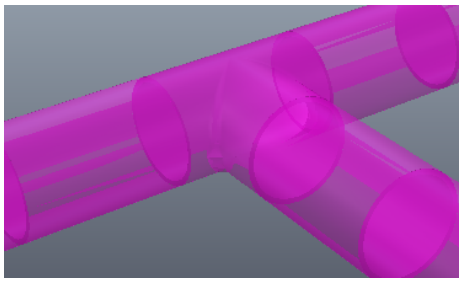
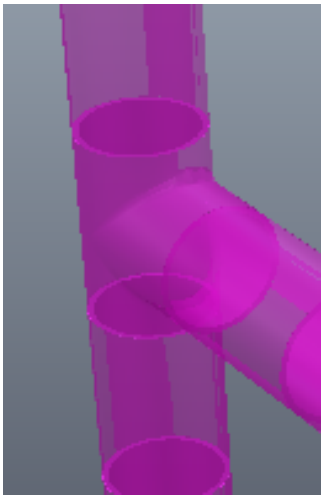


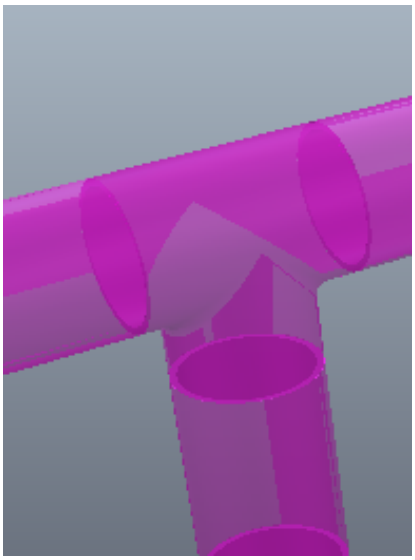
Figure 5.3: Graph of a random pipe network, containing all types of joints



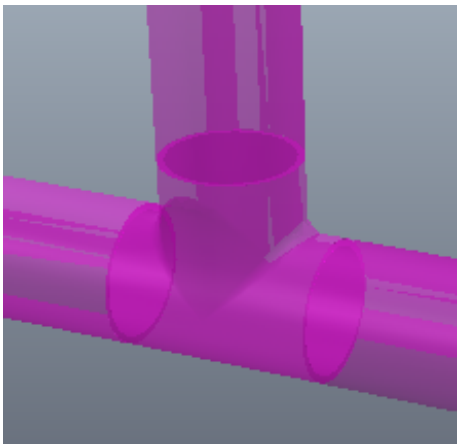
(a) The horizontal-horizontal configuration



(b) The horizontal-vertical configuration



(c) The upwards-oriented configuration



(d) The downwards-oriented configuration

Figure 5.4: Different static pipe configurations used in this project

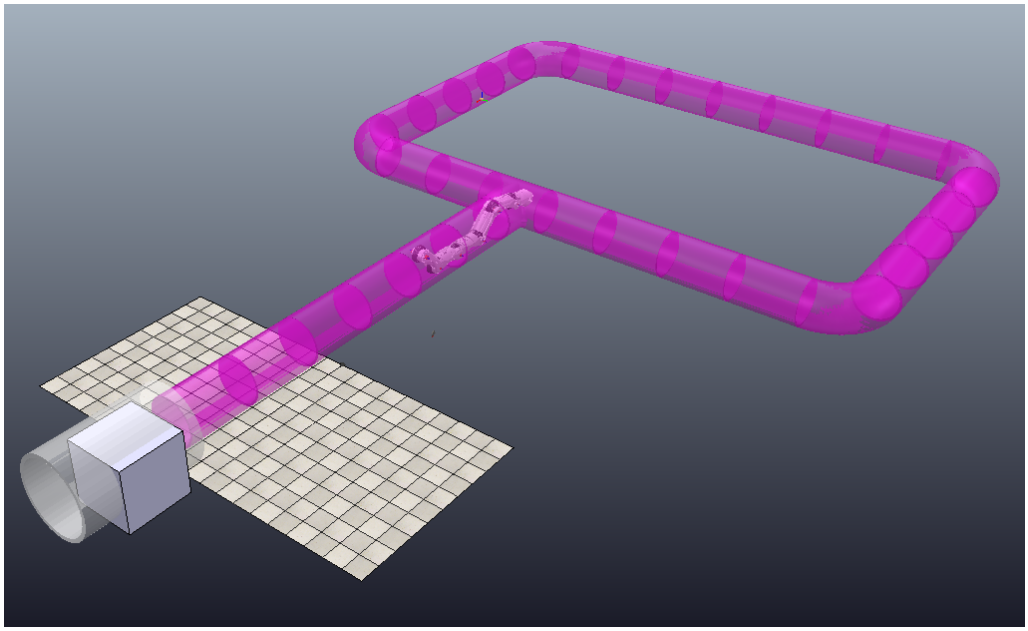


Figure 5.5: Closed pipe network with a single loop

5.1.3 Closing the Pipe Network

Closing the pipe system means that no open ended pipe segments exist in the network, with the exception of the pipe where the robot gets spawned. One simple way to generate such pipe networks is to immediately close the loop when a T-junction is added to the pipe system by joining the two openings of the T-junction with a looping pipe, as seen in Figure 5.5. By branching even more before actually closing the loops, more complex closed network can be obtained. Figure 5.6 shows an example of such a pipe system.

Another way of creating a branching closed pipe network is to create loops that have a common edge, similar to the mesh structure of a net. However, this conformation is not explored in this project, and it will be treated in future work.

5.1.4 Explored Segments

In case of the exploration experiments, presented in Section 4.5, it is important to keep track of the pipe elements which were already reached by the robot. During training, the position of the front of the robot is compared with the position of the pipe elements. The one closest to the robot is marked as being visited, and a metric representing the percentage of the network which was already explored is updated. This percentage is used in the results chapter to analyze the efficiency of the method.

The visitation status of the whole network is recorded in the environment state. Therefore, even if the robot revisits an already visited pipe element, it only gets marked as being visited once, thus triggering a single visitation percentage update. To give a visual representation in the simulator, the visited pipes are colored green, as opposed to the unexplored pipe elements which are colored pink.

5.2 Path Planning

When the PIRATE robot performs a navigation task, it needs to reach a certain end position. Considering the configuration of the pipe network is already known, a path through the branching pipe network can be traced. Using the pipe generation method presented in Section 5.1, information related to the positions of each pipe element and what they are connected to

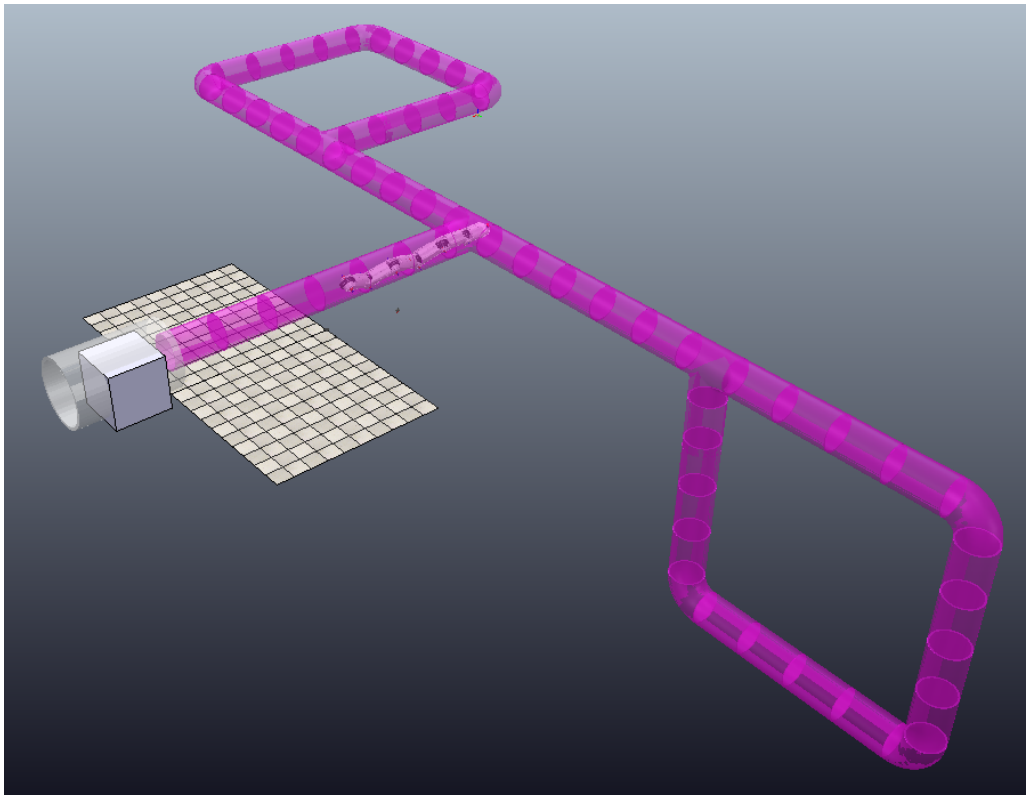


Figure 5.6: Closed pipe network with two loops

becomes available. For simplicity, the path is build between the starting pipe element and the last element added to the network. Starting from the end, a neighboring element is added to the path if its index is smaller than the previous one. In case multiple neighbors exist, the one with the lowest index is chosen. Even if the path obtained may not be optimal, the purpose of this project is to have the robot follow a given path, therefore the optimality of the path is not required.

In Figure 5.7 a path built for the graph at Figure 5.3 can be observed. The red dots represent the waypoints generated to reach the element with the index 15.

5.3 Pipe Environment Setup Considerations

When setting up the pipe network for different experiments, several aspects need to be considered. Firstly, for training agents responsible for navigating through pipe bends, turning inside a corner element is way easier than turning in a T-junction. Secondly, because of the varying pipe diameters introduced by the presence of the reducer element, all the agents should use in their training environments both wider and narrower pipe diameters. This gets translated to an episode having a 50% chance to pick either size, the normal one or the one with a 20% increase, for the diameter of the starting pipe. Lastly, the PIRATE should be able to navigate in any direction, including upwards and downwards, therefore the orientation of the initial pipe needs to be randomized as well.

The following subsections contain particularities for the environment setup for different agents.

5.3.1 Backturn Agent

From all the episodes the backturn agent is ran, 25% of them contain a simple corner pipe element, while the rest use a T-junction element. It should be noted here that it does not matter

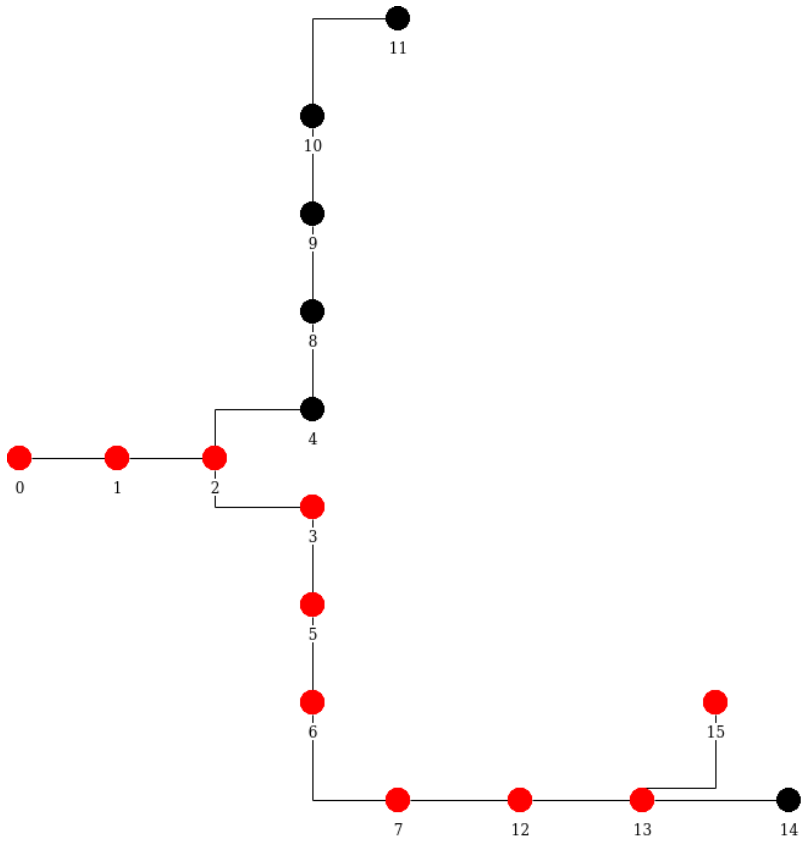


Figure 5.7: Path generated from the start to the node number 15

for the agent if the pipe element is placed transversally or longitudinally with respect to the previous straight pipe segment, as the robot starts the episode already inside the bend, and its conformation is symmetrical. Taking these into consideration, the episodes containing a T-junction still have an equal split between longitudinally and transversally placed pipe elements.

5.3.2 Clamp Drive Agent

In the case of the clamp drive agent, 20% of its training episodes contain a pipe reducer element. Because the drive agent is not concerned with bends in the pipe, the probability values for placing T-junctions can be left to their defaults.

5.3.3 Turn Agent

To ensure that the policy gets trained on a varied environment, the pipe network generation is adapted such that around 75% of the training episodes contain a T-junction. The reasoning behind this is that it is easier for the robot to turn inside a simple bend compared to a T-junction, therefore more training is required for the latter case. As to the placement configuration of the elements, the T-junctions are placed only transversally in later experiments, for reasons further elaborated in Section 6.6.

Later experiments require a static pipe configuration. This means that all the episodes of a particular trial need to use the same pipe network. More information about static pipes is presented in Section 5.1.2.

5.3.4 Pipe Traversal and Exploration

The environment setup for the top level agent uses the default configuration, with a couple exceptions. In the case of the pipe traversal experiments, the setup is similar to the turn agent

Parameter	learning rate	gamma	lambda	clip	entropy coeff
Range	{5e-6, 1e-5, 5e-5, 1e-4, 5e-4, 1e-3}	[0.7, 0.9997]	[0.7, 1.2]	[0.1, 0.3]	[0.001, 0.02]

Table 5.1: The tuning ranges for the parameter tuning experiments

agent, except that only 50% of the bends are T-junctions, and all of them are transversally placed.

In the case of the exploration experiments, the pipe network should be closed. This is to prevent the robot from falling from the pipe when exploring new segments. Because closing the pipe network implies creating loops, all of the bends placed in by the algorithm outside of the loops are T-junctions. From these, 75% of them use a transversal configuration. The reason for this is to increase the variety of the environment: if the first pipe element has a longitudinal conformation, the pipe generation algorithms creates a single loop network. However, a transversally placed one produces more varied environments, with either one or two loops. Simple bends are not of interest for this approach, as the focus falls more on exploring the environment than being able to navigate through simple pipe corners - which is performed while navigating the pipe loops. Section 5.1.3 shows examples of closed pipe networks.

5.4 Population Based Training

Hyperparameter tuning is an important part of building a robust ML model. In the case of the PIRATE navigation controller, the end goal is to be able to deploy the final model on the real system. Therefore, it is even more crucial to have a well tuned model.

The parameters tuned for this project, as well as their tuning ranges, are shown in Table 5.1. Here, *learning rate* is the learning rate of the agent (also abbreviated *lr*), *gamma* the reward discount factor, and the rest are hyperparameters specific to the PPO algorithm. All the parameters except the learning rate are sampled from a continuous uniform distribution in the range shown in the table. The learning rate on the other hand is sampled uniformly from a discrete search space.

The tuning method used in this project is PBT. Multiple training processed are initiated in parallel, each one using a different hyperparameter combination. Periodically, namely every 50 iterations, the models are compared using the mean reward per episode as a metric, and ones performing worse have their hyperparameter values, as well as their weights replaced with the ones of the models that have a better performance. Furthermore, the values of the parameters are mutated as well with a certain probability, trying new hyperparameter combinations. For this reason, it is important to check the progress of all the agents under training when picking the final parameter values, as an intermediary agent may perform better than the ones at the end of the tuning process.

The tuning process is repeated for each subpolicy: the backturn agent, the clamp drive agent and the turn agent. The results and observations of the tuning are presented in Section 6.2.

5.5 Summary

In this chapter the setup of the experiments are presented. Section 5.1 treats the procedural algorithm that generates a pipe network suitable for the experiments. Depending on the requirements, the probabilities of different pipe elements can be configured, thus obtaining all kinds of pipe networks of different sizes and shapes. Afterwards, Section 5.2 shows how a path can be constructed from the generated pipe network. Using a backtracking approach, a path containing intermediate waypoints of all the pipe elements that have to be traversed is built from the start of the pipe towards the pipe element with the highest index - namely the last

one added to the network. Next, Section 5.3 describes how different experiments use different configurations of the pipe network. Finally, Section 5.4 tackles the hyperparameter tuning performed before the actual experiments, showing which parameters are treated. The tuning process uses a Population Based Training method to find the right parameters for each agent.

6 Results and Discussion

This chapter presents the results of the experiments described in Chapter 4. The experiments are setup using the methods shown in Chapter 5. The results are then analyzed, discussing how well the methods used are able to answer the research questions.

6.1 Metrics

The results are presented as graphs and tables employing different metrics. One of the metrics used to evaluate a policy is the mean reward per episode, representing the average reward obtained by the agent during training across all the episodes of an iteration. The information this metric conveys is how well a particular agent behaves in comparison with other agents trained on the same environment, or how much the policies improve during training.

To get a better idea of how well an agent behaves in an absolute manner, the number of successful episodes with respect to the total number of episodes ran in a training scenario is considered. The slope of this graph is a number between 0 and 1 indicating the number of episodes where the episode finished successfully. Because the graph of successful episodes with respect to the total episodes does not provide all the information (as it is in most cases a simple line), and the slope of this graph is noisy due to the random nature of the environment, the average success rate for all the episodes until the current iteration in a certain window size is used instead to plot the performance. This acts similarly to an averaging filter and showing how the agent gets improved in performing its task, as well as its final performance. It is also relevant to consider the total average performance over the training - namely the number of successful episodes divided by the total episodes at the end of the training.

Similar to the success rate, in the case of evaluating the turn agent performance the success rate for reaching the is considered as well. In this context, this metric represents the number of times the robot reached the wrong target. Comparing the success rate for reaching the right target with the success rate for reaching the wrong target yields important information, such as how well can the agent choose between the two possible options, as well as highlighting the rate of which the robot gets stuck in the T-junction.

Where appropriate, the evaluated metric becomes the distance travelled by the robot, or the degree of exploration of the pipe network. The former is relevant for agents that focus on moving the robot forward, such as the clamp drive agent, while the latter is used for showing the performance of the top level agent responsible for pipe exploration.

6.2 Hyperparameter Tuning

Previous work tuned the policies using a grid search method. The resulting hyperparameters (called the previous work parameters), along with the parameters obtained during this project using PBT, are shown in Table 6.1.

Parameter	learning rate	gamma	lambda	clip	entropy coeff
Previous Work	5e-5	0.99	0.95	0.2	0.01
Backturn Policy	0.001	1.14	0.008	0.24	0.792
Clamp Drive Policy	5e-5	0.99	0.95	0.2	0.01
Enter Turn Policy	1e-5	0.9	1.1	0.24	0.0025

Table 6.1: The hyperparameters obtained after tuning the models on different scenarios

	Before Tuning	After Tuning
Backturn Agent	55%	69.3%
Turn Agent	32.2%	35.8%

Table 6.2: Comparison between the success rate for the tuned agents

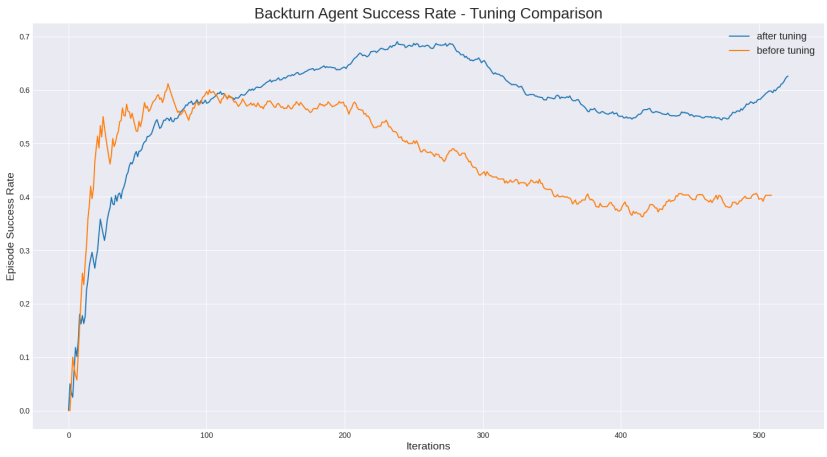


Figure 6.1: Comparison between the backturn policy model before (orange) and after tuning the hyperparameters (blue)

The parameter combination is obtained from observing the mean reward per episode for the models being trained, and picking the ones associated to the best performing model. It is assumed that the best performing agent is the one that has a higher reward per episode than the rest of the agents, across the perturbations during training. To test whether the parameter tuning actually obtains models performing better, one model is trained with the old hyperparameters and another one with the tuned ones. Their performance are compared using different appropriate metrics. The following subsections treat each tuning experiment separately. Table 6.2 shows the success rate comparison between the agents trained with the hyperparameters before and after tuning.

6.2.1 Backturn Policy Tuning

The average reward per episode for the whole tuning process is shown in Figure A.1. It can be seen that the agent corresponding to trial ID 02 outperforms the other agents between iterations 300 and 350. The corresponding parameters of this agent are placed in Table 6.1.

To show that an agent using the new hyperparameter combination performs better than the old one, two agents are trained using the old and new parameters. The comparison is shown in Figure 6.1. It can be seen that the tuned agent performs better than the one using the parameters before tuning, in terms of the average success rate.

6.2.2 Clamp Drive Policy Tuning

Similar to the backturn policy, the full tuning progression is shown if Figure A.2, and the parameters corresponding to the tuned agent are extracted and filled in in Table 6.1. In this case, the trial ID 06 outperforms the others, between iterations 200 and 250. Checking the hyperparameter combination of this agent, it is remarked that they coincide with the initial parameters. Therefore, the tuning process did not manage to obtain a hyperparameter combination that yields an agent with better performance than the baseline.

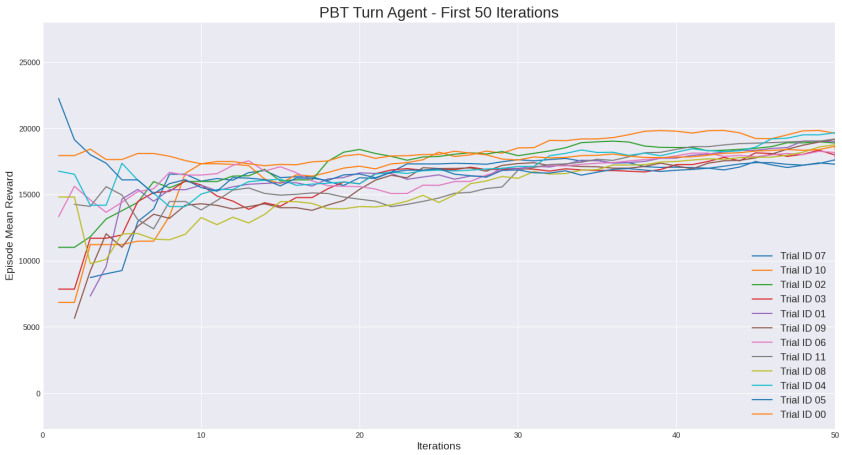


Figure 6.2: PBT training for the turn policy, first 50 iterations

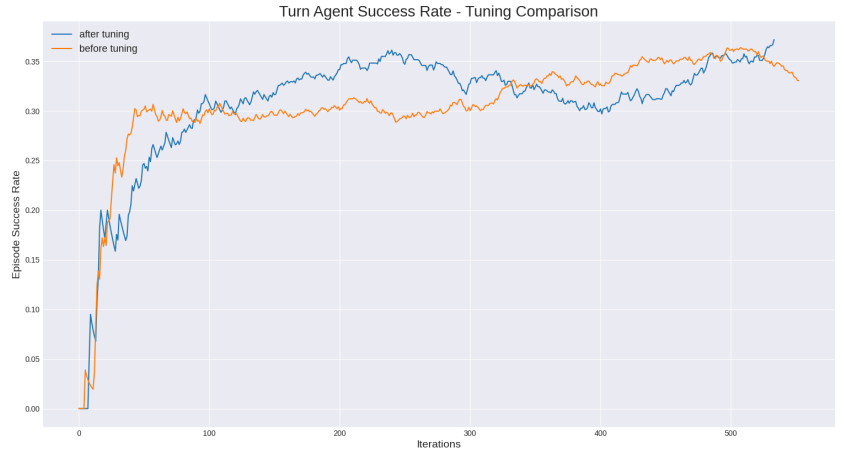


Figure 6.3: Comparison between the turn policy model before and after tuning the hyperparameters

6.2.3 Turn Policy Tuning

A sample of the training results is shown in Figure 6.2. This graph shows the reward for the first 50 iterations of the 12 agents being trained in parallel. The graph for the whole training is placed in Appendix A. Checking the whole graph in the Appendix, Figure A.3, the chosen turn agent is the one with the ID 10, due to its performance between the 250 and 300 iterations. From this final graph, the parameters for the agent having the best performance are added into Table 6.1.

Figure 6.3 shows the difference in the success rate for a model trained with the old hyperparameters and the ones tuned specially for this scenario. As it can be seen, during training the tuned model did perform better than the not tuned counterpart. The performance difference is 35.8% for the tuned agent, and 32.2% for the one trained with the old parameters.

6.3 Backturn Policy

The backturn policy acts when the robot has already entered a pipe bend or T-junction and needs to leave. Figure 6.1 already shows the mean reward per episode obtained by the tuned

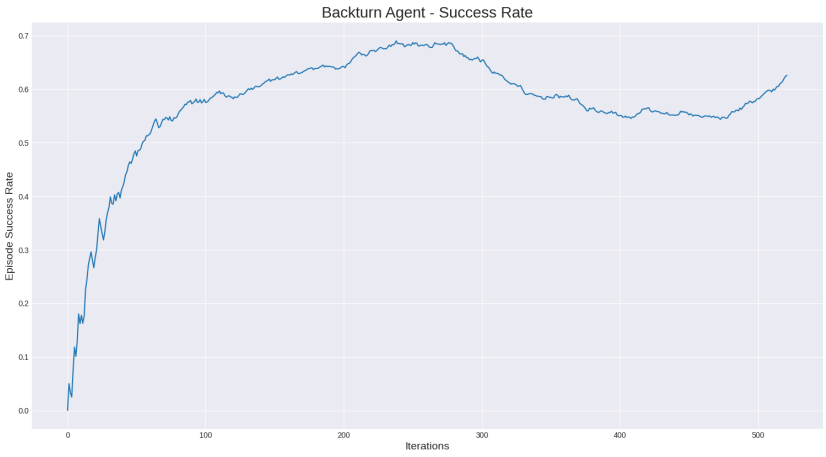


Figure 6.4: The average success rate for the backturn agent

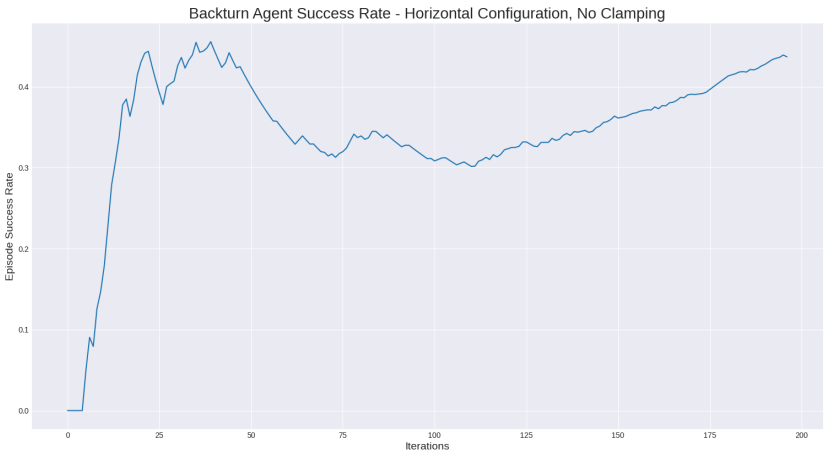


Figure 6.5: The average success rate for the backturn agent in a static environment, without the forced clamping action

agent over the training iterations. As the graph shows, the agent accumulates more reward as the training progresses.

Furthermore, Figure 6.4 shows the progress of reaching the end target during training. At the end of the training, the agent manages to reach the target in around 70% of the total episodes. This means that the agent did manage to perform better during training. While this number is not as high as it can get, running the subpolicy alongside the top level agent can improve the result. As the top-level agent employs other subpolicies, for instance the clamp driving one, the robot is still able to advance in situations that the exit turn subpolicy is not able to cover. Whether this statement proves to be true or not is tackled in Section 6.7.

To test whether the agent is able to successfully leave a T-junction, the trained agent is tested on static horizontal T-junctions. The first remark is that, because of the forced clamping, the robot slips back inside the turn, thus not managing to reach the desired position. Therefore, the initial high success rate can be attributed to simple pipe bends and downwards-facing pipe configurations, where the gravity helps the agent perform its task.

Testing again on a horizontal configuration, but this time without the forced clamping action, the agent managed to achieve a success rate of 47%. The success rate can be seen in Figure 6.5. While this performance does not guarantee a proper T-junction traversal, it is possible that this trained policy is enough when integrated with the whole hierarchical structure.

6.4 Clamp Drive Policy

The clamp drive subpolicy is responsible for navigation through straight pipe segments. The agent is trained in pipe networks with varying diameter of the pipes, as well as an added pipe diameter reducer element.

The performance of the training of the RL model on the new pipe configuration is shown in Figure 6.6. The graph in Figure 6.6a shows the average reward per episode with respect to the training iterations. Just by observing this graph, it can be seen that the reward increases with the iterations. Furthermore, Figure 6.6b shows an increasing trend in the distance travelled each iteration. This distance plot is obtained by taking the actual distance and applying an average filter on the samples of length 20.

The increasing trend shown in Figure 6.6b shows that the policy manages, as it gets trained, to travel more distance each training iteration. Also, looking at Figure 6.6c it can be seen that the agent manages to reach the end goal on average in over 90% of the training iterations, having an overall success rate of 97.7%. This result shows that the policy gets trained successfully, increasing its success rate over time, and it manages to easily adapt to different pipe diameters and orientations.

6.5 Rotational Policy

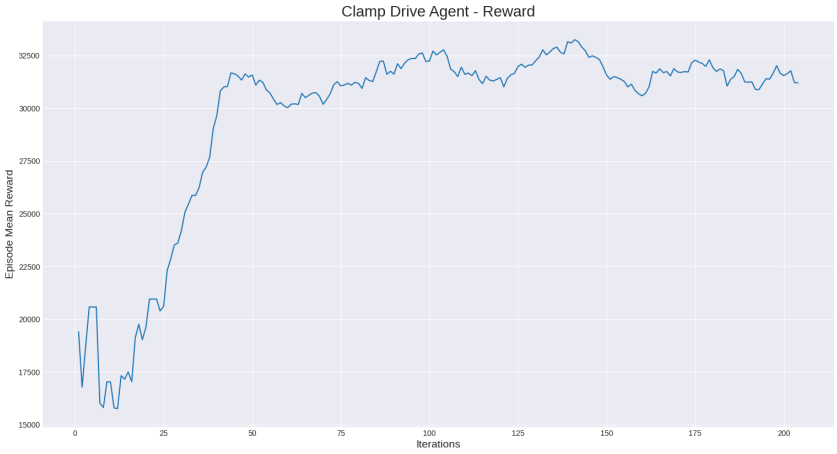
The rotational agent is firstly trained with only joint positions. However, the experiments show that the robot does not properly clamp, which also means that the agent cannot rotate the robot in place in case of vertical pipes. To encourage the robot to clamp, a second approach uses a different action space, with the second joint receiving a continuous rotational velocity. The evaluated metrics for this scenario is shown in Figure 6.7.

As it can be seen in the reward graphs, the mean episode reward does not increase over the training time. The robot does clamp because of its forced clamping action, therefore preventing falls through vertical pipe segments. However, looking at Figure 6.7b, the average success rate of the agent is around 50%, with a total of 57% of episodes being successful. However, this is only a partial rotation, as the robot only rotates half of its body. In this case, the expected success rate should be way higher to ensure that the robot can rotate properly inside the pipe. The low performance can also be attributed to the symmetric nature of the pipe, as the camera information cannot offer reliable information about the relative orientation of the robot inside the pipe.

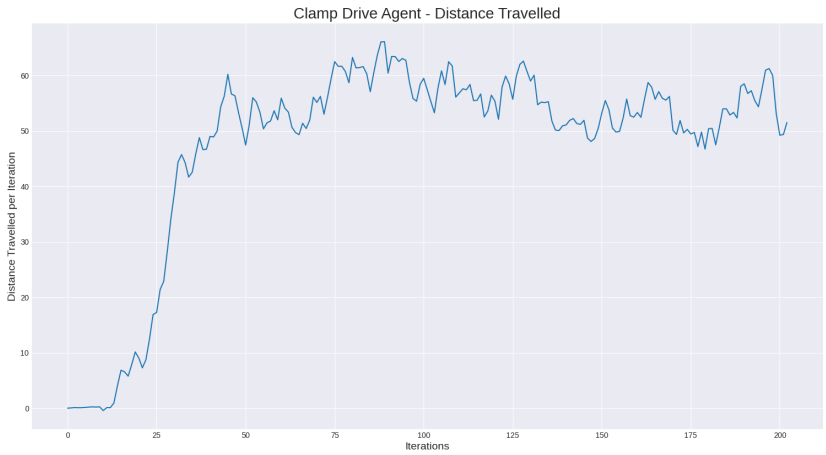
One solution to having the robot rotate properly is to mimic the action pipeline observed using manual control: clamping one side of the robot while rotating, and then finishing the rotation with the other side clamped. The method proposed in this project does not manage to accomplish this complicated task. The HRL structure can be updated with an intermediate master rotation policy, with its own subpolicies. However, this leads to a very complicated structure for the navigation controller, and therefore is not considered in the project.

6.6 Enter Turn Policy

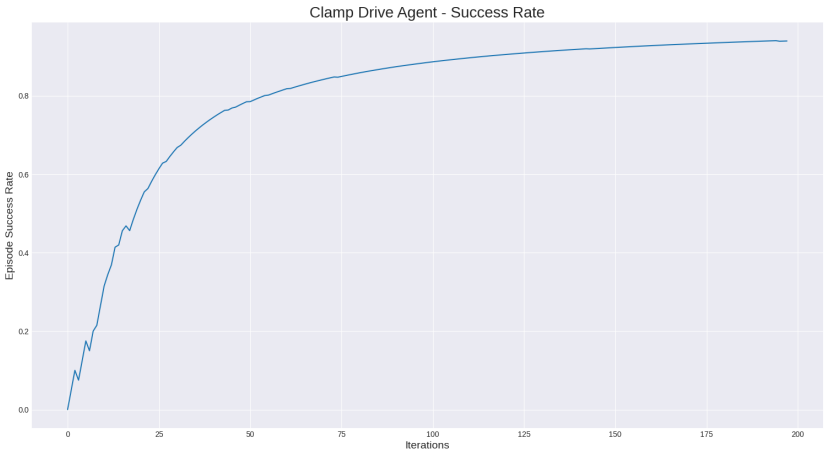
This section treats the results of the experiments that aim to achieve an RL agent able to navigate through both T-Junctions and pipe corners.



(a) The mean reward per episode

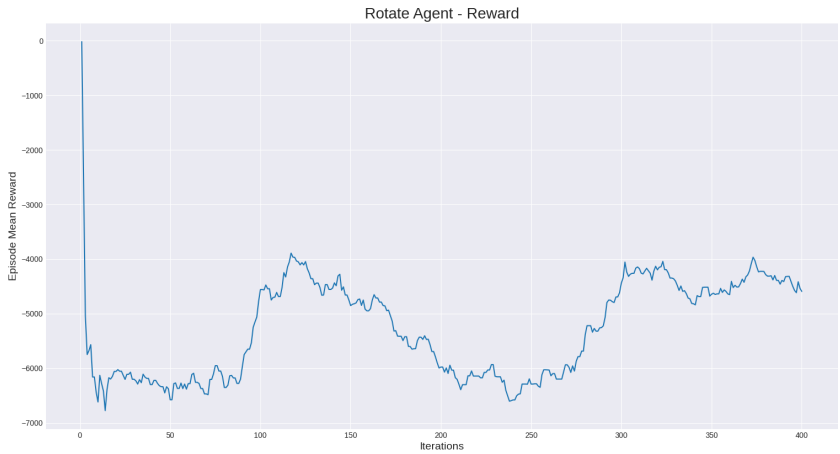


(b) The distance travelled

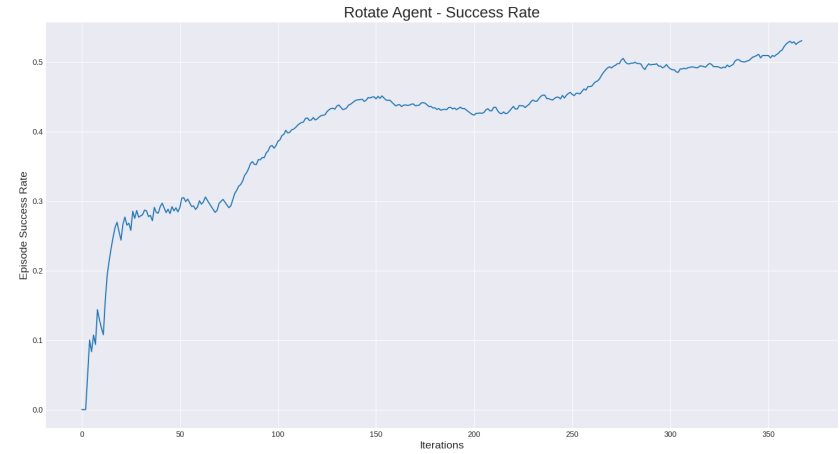


(c) The episode success rate

Figure 6.6: The results of training the clamp drive subpolicy



(a) The mean reward per episode with respect to the iteration



(b) The number of times the goal was reached with respect to the total episodes

Figure 6.7: The results of training the rotation policy with forced clamping actions

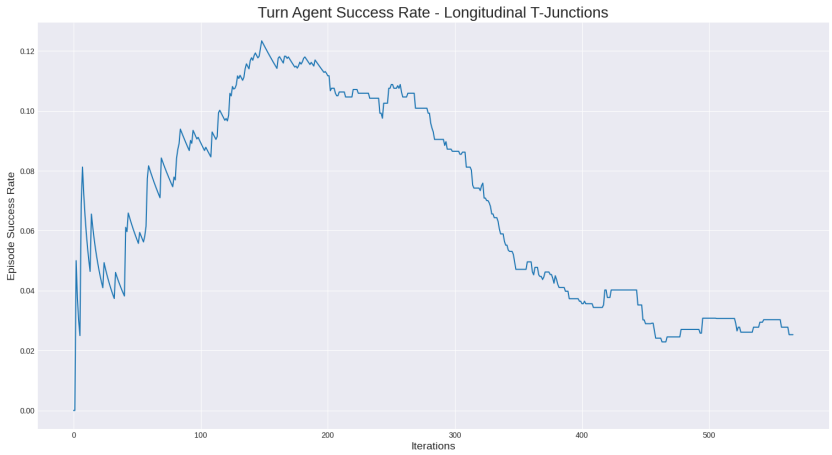


Figure 6.8: The success rate of the turn agent trained on longitudinal T-junctions

6.6.1 Longitudinal Turns

It has been remarked that while navigating the longitudinally placed T-junctions, the trained policy does not attempt to take the turn. This is also proven in Figure 6.8. In the graph, the success rate over the iterations is plotted for an experiments that retrains an already trained agent only on longitudinal T-junctions. The experiment shows that the success rate for this scenario is around 6%, which is low enough to be attributed to randomness - for instance the episodes with the bend facing downwards, when the robot can fall through.

One cause for this can be the difficulty of distinguishing the bend from the camera observation. Because of this, the agent is not able to identify the bend, and continues driving forward. To avoid this behaviour, only turns in transversally placed T-junctions are considered. Entering turns in a longitudinal tee element is left to be considered in future work.

6.6.2 Action Space, Policy Model and Subgoal Variations

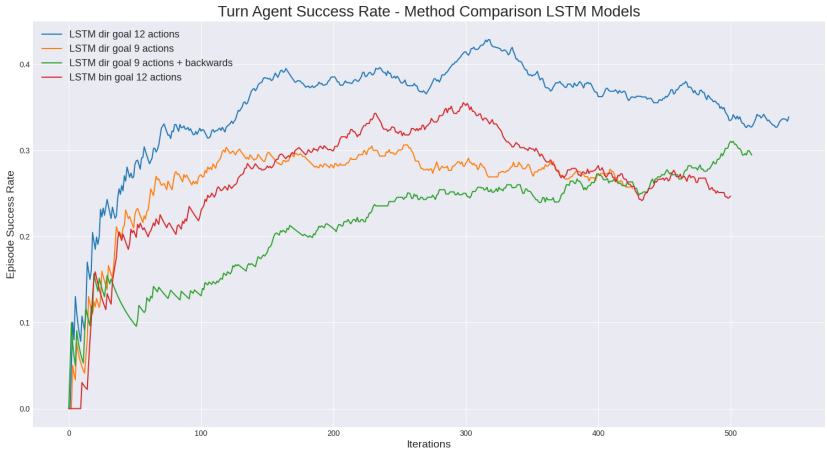
Figure 6.9 shows the rate of successful episodes with respect to the number of total episodes for different approaches, as a function of the current iteration. On the raw results, an averaging filter of length 100 is added, as the performance of the older iterations is no longer relevant for evaluating the fully trained model. The methods considered use either a CNN or an LSTM network for the policy, a directional (dir) or binary (bin) goal, and an action space of size either 9 or 12 - the difference being whether the front wheels rotate or not. Furthermore, agents that can or cannot drive backwards are considered as well. The full breakdown of the compared models and their performances is presented in Table 6.3. The numbers presented in the table represent the overall performance over the whole training process.

The best performing agent is the one using a recurrent network, directional subgoal and full wheel control. Furthermore, the agent that can steer backwards does also not succeed in achieving the goal in a high number of episodes. The reasons for this can be that the agent cannot handle driving in vertical pipe segments, as it stops clamping, as well as not being able to decide when to drive forward or backward.

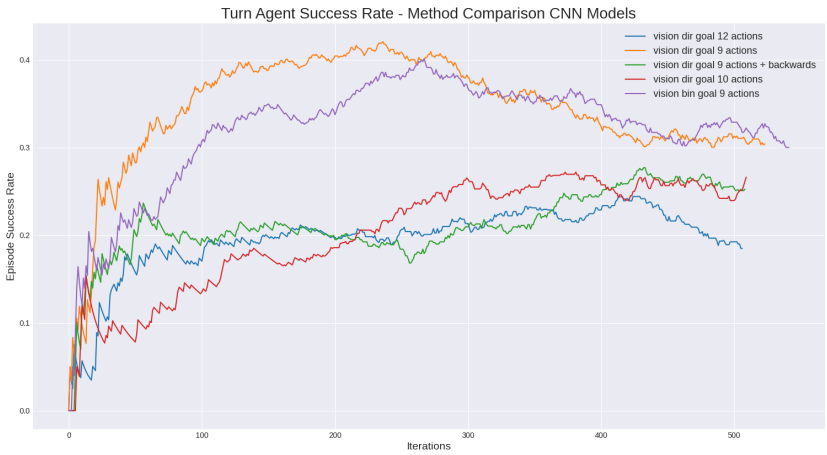
From Table 6.3 it can be remarked that in the case of the CNN policy, the best approach is to use 9 degrees of freedom action space, with forced forward movement. The difference between using a directional goal or a binary goal is not substantial showing an overall performance of 37% and 35% respectively. In the case of the recurrent policy, the agent manages to reach the target in a total of 40% for an extended action space and 35% for the partial wheel control.

Model Used	Goal Type	Size of Action Space	Performance [%]
LSTM	directional	12	40%
LSTM	directional	9	35%
LSTM	directional	9 with backward possibility	26%
LSTM	binary	12	30%
CNN	directional	12	20%
CNN	directional	9	37%
CNN	directional	9 with backward possibility	24%
CNN	directional	10	25%
CNN	binary	9	35%

Table 6.3: The efficiency of the methods tried for navigating through pipe junctions, measured in the percent of successful episodes



(a) Using the recurrent policy



(b) Using the CNN policy

Figure 6.9: Comparison of successful episodes for different approaches of the turn agent implementation

Pipe Configuration	Performance [%]
Horizontal-Horizontal	36.5%
Horizontal-Vertical	44.6%
Upwards	0%
Downwards	31.8%
Longitudinal	5.9%

Table 6.4: The efficiency of the agent trained on different static configurations, measured in the percent of successful episodes

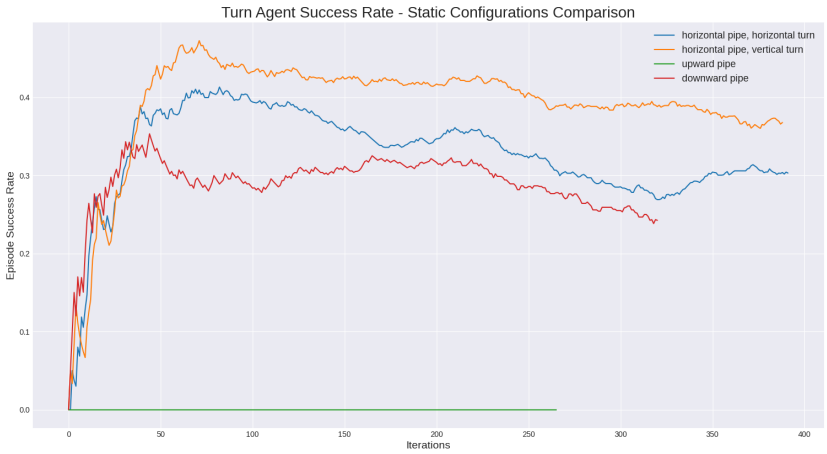


Figure 6.10: The success rate breakdown for the four possible T-junction configuration

6.6.3 Static Configuration Comparison

Because of the relatively low success rate of the methods tried, a performance breakdown into the different pipe configurations is needed to assess for which orientations of the T-junction the agent struggles the most. Four possible configurations are tested: a horizontal straight pipe segment, followed by a horizontally (respectively a vertically) oriented T-junction; a straight pipe segmented oriented upwards; and a straight pipe segment oriented downwards. The agent used for these experiments is the one using a directional goal, an action space size of 9, and a CNN model for its policy.

Figure 6.10 shows the average success rate with respect to the number of iterations for the four possible configuration. It can be seen that the agent is not able to reach the goal in any of the episodes for the vertical configuration. The total performance for these experiments is shown in Table 6.4.

6.6.4 Curriculum Learning

It is apparent from the previous section that a direct approach is not very efficient for achieving a high performance. Therefore, a curriculum learning method is employed. Furthermore, the policy model size is also extended, as described in Section 4.3.5. In the first step, the previous model is compared with the new one. Also, the robot is spawned inside the T-junction, facing the goal. Figure 6.11 shows the comparison between the old model and the new one for this scenario. It can be seen that the agent can solve this trivial case way better with an extended model, compared to the initially considered one - the difference between 97.7% success rate and a 90% one. This means that the next steps of the curriculum learning are performed with the better performing model.

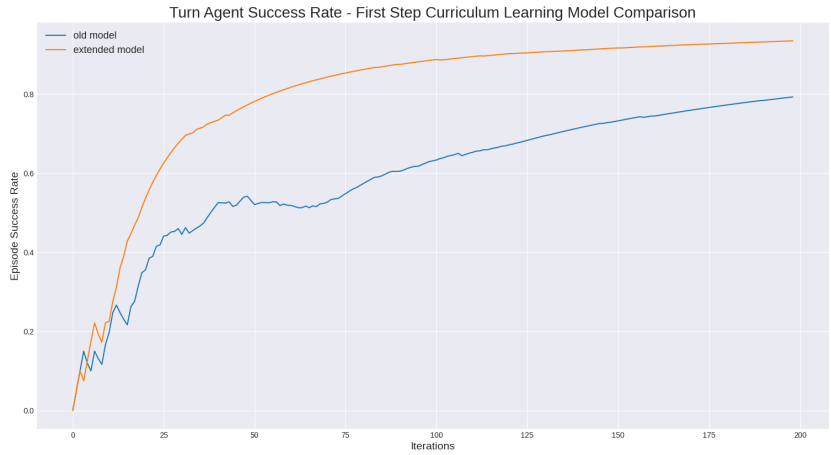


Figure 6.11: Model size comparison for the first step of the curriculum learning

Method	Success Rate (Correct Target) [%]	Success Rate (Wrong Target) [%]
Directional Goal	48.3%	45.5%
Binary Goal	75%	0.6%
Position Goal	48.7%	43.7%

Table 6.5: The overall success and failure rate for the second step of the curriculum learning for the turn agent

During the second step, the success rate and the success rate for reaching the wrong target are compared. In this case it is important to highlight how well the agent can use the goal information. Table 6.5 shows the overall success rate and failure rate for different approaches. In this case the binary goal experiment is performed with an observation of 1 and 2 instead of the positive or negative angle difference. The graphs showing the performance for these three models can be seen in Section B.

These results show that the agent cannot distinguish between a left and a right turn using the position goal and the directional goal. Even if the right target is reached in more cases compared to the wrong one, the difference is not high enough to be considered a success. However, the agent manages to reach one position or the other in most of the cases, meaning that it does not get stuck in the T-junction. The possible reason that the binary subgoal agent manages to reach the right target and the others do not might be the shape of the goal itself: the binary goal is a single value, while the others are a 3-dimensional vector. It is possible that the current shape of the policy model is not suitable for a vector-shaped goal.

In comparison, the agent using the binary subgoal reaches the right target in most of the cases, but it gets stuck more often. It is possible that the cases then the robot remains stuck can be ameliorated after integrating the agent in the hierarchical structure.

The third step of the curriculum learning approach means moving the right target further away from the center of the T-junction. Unfortunately, the agent is not able to do that, as the robot tends to get stuck. The reason for this is the same one presented in Section 6.7.

6.7 Top Level Traversal

To integrate the decision-making of the turn policy with the other two agents, a top level agent is trained. The challenge is to navigate through a pipe network with a horizontal T-junction, and reach the end of the pipe segment after the turn. Testing the trained model, it is observed

that the actions of the backturn policy manage to get the robot unstuck when it gets blocked inside the T-junction. This fact covers the cases where the turn agent is not able to properly enter the bend properly. However, the top level agent suffers from the same issue as the backturn agent with forced clamping. Namely, clamping the front of the robot results in the PIRATE slipping back inside the T-junction. The action itself should not be removed, as is it still required for properly leaving the turn after most of the PIRATE passes through the junction. This issue shall be addressed in future work.

6.8 Top Level Exploration

To properly test the exploration, it is necessary to firstly obtain a high success rate of the traversal. Unfortunately this is not achieved in the scope of this project, and is considered for future work as well. However, it is important to note that the setup and logic for performing these experiments are already implemented through the work invested in this project.

6.9 Summary

Performing PBT before doing experiments on different policies lead to agents whose hyperparameter combination result in better agent rewards. These parameters are shown in Table 6.1. The performance of the tuned agents is compared to the performance of each equivalent agent, trained with the old parameters.

After the tuning, the subpolicy level agents are trained on the new environment. First are the clamp drive agent and exit turn agent. Afterwards, multiple approaches are tried to adapt the navigation controller to navigate through both simple pipe bends and T-junctions. The approaches are employing a combination of the options framework with the feudal learning, employing an subgoal in the HRL architecture. While a rotational subpolicy approach proves ineffective, the focus is shifted on adapting the turn agent to be able to take the turn decision.

The different methods tried in this project on the turn agent are compared using the episode success rate. Firstly the generalization case is tackled, considering different combinations of action spaces and models. As the results do not show a high success rate, the problem gets simplified. The first simplification implies comparing the different T-junctions configurations using the agent that had the best performance in the generalization experiment. Afterwards, the horizontal pipe configuration is further explored by performing curriculum learning, investigating the effect of different models and observation space variants on the performance.

Finally, the traversal experiments prove unsuccessful due to the front clamping dragging the PIRATE back inside the T-junction. Without a high performance on the traversal side, the exploration experiments are left as future work.

7 Conclusion

In this chapter the answers to the research questions posed in Chapter 1 are formulated, based on the results from Chapter 6. Furthermore, ideas and recommendations for future work are exposed.

7.1 Answers to the Research Questions

The aim of this project is to explore ways to extend the current functionality of the PIRATE robot, such that it is able to navigate through more complex pipe elements. This includes pipe reductions and T-junctions. Moreover, a more complex tuning procedure than the one used in previous work is explored, in order to attempt increasing the performance of the RL agents.

To what extent can the current HRL framework be adapted to support pipe diameter changes and navigation through T-junctions?

From the experiments it can be seen that the robot is able to clamp properly inside the pipes, and can drive forward inside the pipe network, because of the forced clamping action. Navigating through the reducer pipe element is easily accomplished by the clamp drive agent, as seen in Section 6.4.

Regarding T-junctions, the chosen approach for this project relies on a combination between the options framework and the feudal learning methods. Adding subgoals to one or more of the subpolicies leads to a navigation controller able to take decisions. A rotational subpolicy that tries to orient the front half of the robot according to a setpoint angle proves to be ineffective. Changing the approach, the turn agent is modified to accept a subgoal that dictates which of the two possible actions to pick. In case of a longitudinal T-junction, the agent fails to reach the target placed in the bend. In case of a transversal one, it is able to enter the junction without getting stuck, and take the right decision in most of the cases.

What decision model for T-junctions can be used to ensure full exploration of the pipe network?

Two main decision models are considered in this project. The first one employs a binary decision, dictating whether the robot should take a left turn or a right turn. Depending on the initial orientation of the robot, this can prove to be ambiguous, because of the "antisymmetry" of the problem. In other words, a left turn becomes a right turn when the robot gets rotated by 180 degrees.

The second approach for the decision model is considering an absolute frame of reference and considering 6 directions in space, one for each half-axis of the three-dimensional space. The decision becomes harder to make, but it solves the relative orientation ambiguity.

The binary approach proves to be efficient in most of the cases, as the agent manages to choose the right turn. Whether ensures a high performance for exploration missions remains as future work.

How can the HRL model be extended to support different types of inspection missions (e.g. exploring a branching pipe network, following predefined paths)?

The top level agent works with one of the two possible scenarios. In the first case, the pipe network is considered to be known, and the waypoints of a path in the pipe network leading to the desired final position can be calculated. Reaching one of such waypoints offers a reward to the agent, while straying too far from the path (for instance taking a wrong turn in a T-junction) penalizes it.

In contrast, for an exploration mission the individual decisions are not that important. The exploration progress is recorded in the environment and is used to reward the agent. In this case, there are no target positions or waypoints, therefore the agent cannot rely on this information for navigation purposes.

Traversing a T-junction proves to be a difficult task. Even if the agent is able to choose the right direction, it tends to get stuck inside the pipe because of slip caused by the front clamping action. More research is required to achieve this task.

Can a population based training approach be used to increase the robustness of the PIRATE navigation controller model, compared to the present implementation?

The three subpolicies are tuned using PBT. In two out of the three, a better performance is obtained compared to the model tuned previously using grid search. The average reward per episode shows an increase even with a relatively low number of parallel trials. However, it should be noted that due to the highly random nature of the method, increasing the number of trials would also increase the chance of obtaining a model that performs even better.

7.2 Future Work

As mentioned above, navigating a T-junction while preventing the robot to slip backwards remains a challenge that should be tackled in future research. Besides that, while the current project tackles transversal T-junctions in more depth, navigating through the bend in a longitudinal junction is a complicated task. The camera information within the current navigation controller framework might not be enough to assess whether the robot is placed in such a pipe element or not. This is an aspect that needs to be approached in future work.

Furthermore, being able to just navigate an unknown pipe network is not enough for exploration missions. The navigation has to be coupled with a mapping function, for instance by employing a SLAM algorithm. This offers additional information to the robot, which is crucial for running the robot outside a simulation environment.

In the current implementation, there is no fallback mechanism in case the robot gets stuck inside a pipe element. By adding this feature, the robot would be able to drive backwards and try again. Using also feedback information related to the current position of the robot, the PIRATE can try to undo a previous decision in a T-junction which proves to be wrong.

A PBT Results

The whole graphs of the mean reward per episode with respect to the training iteration are placed in this Appendix. Figure A.1 shows the tuning process for the backturn agent tuning, Figure A.2 for the clamp drive agent, and Figure A.3 corresponds to the enter turn policy tuning.

All of the graphs show the progression for all of the trials running in parallel, identified by their ID (a number from 0 to the total number of parallel trials). Every 50 iterations a parameter update is performed on the worse performing agents.

The corresponding parameters for the best performing agents are already recorded in Table 6.1.

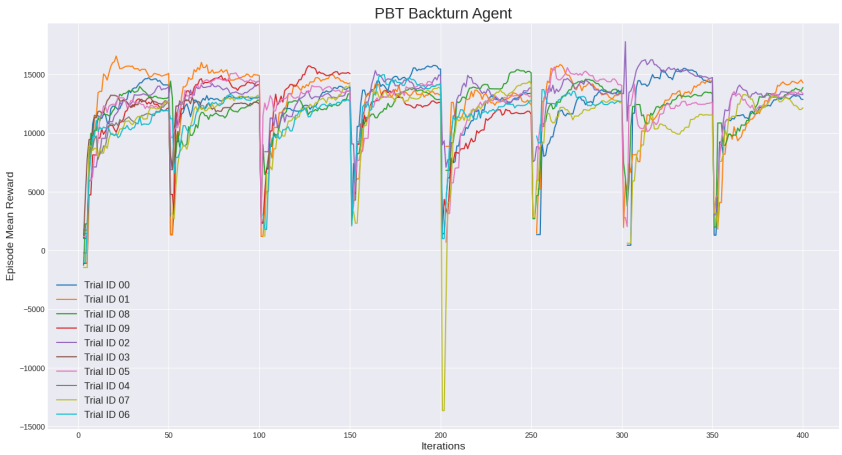


Figure A.1: The full PBT results for the backturn agent tuning

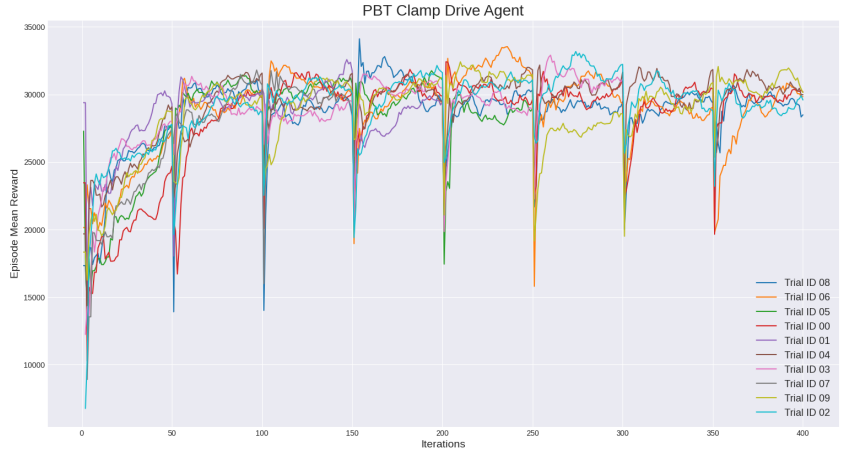


Figure A.2: The full PBT results for the clamp drive agent tuning

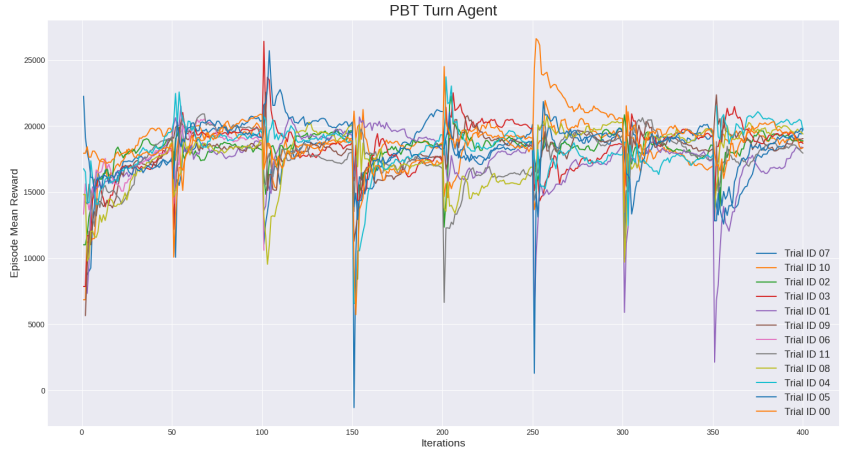


Figure A.3: The full PBT results for the turn agent tuning

B Curriculum Learning Goal Variations

The graphs presented here show the performance of the agents trained with the second step of the curriculum learning approach. Three approaches are considered: an agent using a directional goal, one using a binary goal, and one using a position goal. The graphs contain the success rate of reaching the correct goal, as well as the success rate for reaching the wrong goal. Figure B.1 shows the performance of the agent using the directional goal, Figure B.2 the binary goal, and Figure B.3 the position goal.

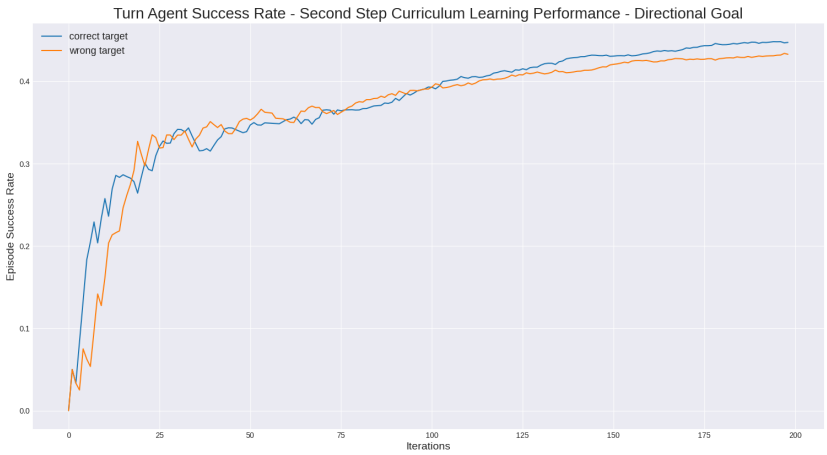


Figure B.1: The performance of the curriculum learning, second step using a directional goal

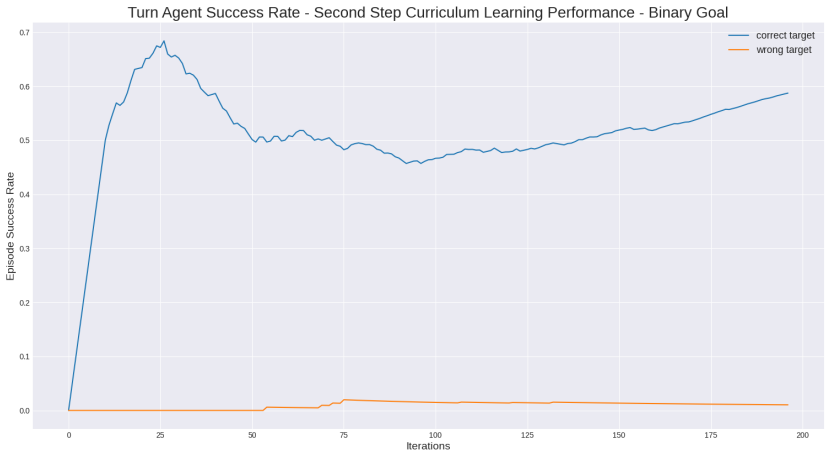


Figure B.2: The performance of the curriculum learning, second step using a binary goal

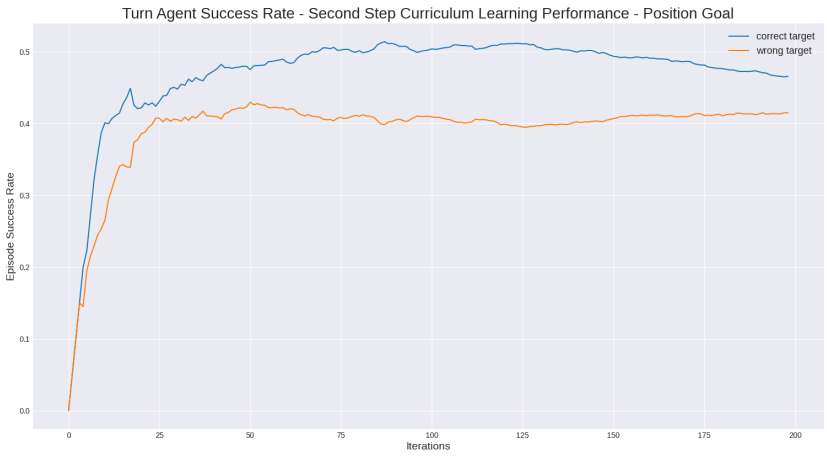


Figure B.3: The performance of the curriculum learning, second step using a position goal

Bibliography

Degris, T., M. White and R. S. Sutton (2012), Off-policy actor-critic, *arXiv preprint arXiv:1205.4839*.

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.

Flet-Berliac, Y. (2019), The Promise of Hierarchical Reinforcement Learning, *The Gradient*.

Garza Morales, G. (2016), *Increasing the Autonomy of the Pipe Inspection Robot PIRATE*, Master's thesis, University of Twente, Netherlands.
http://essay.utwente.nl/71300/

Grefte, L. (2020), *Autonomous navigation of the PIRATE using reinforcement learning*, Master's thesis, University of Twente, Netherlands.
http://essay.utwente.nl/83150/

Jaderberg, M., V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan et al. (2017), Population based training of neural networks, *arXiv preprint arXiv:1711.09846*.

Kahn, G., A. Villaflor, B. Ding, P. Abbeel and S. Levine (2018), Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation, in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 5129–5136.

Kim, W., A. Kanezaki and M. Tanaka (2020), Unsupervised Learning of Image Segmentation Based on Differentiable Feature Clustering, *IEEE Transactions on Image Processing*, **vol. 29**, p. 8055–8068, ISSN 1941-0042, doi:10.1109/tip.2020.3011269.
http://dx.doi.org/10.1109/TIP.2020.3011269

Kulkarni, T. D., K. R. Narasimhan, A. Saeedi and J. B. Tenenbaum (2016), Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation, *arXiv:1604.06057*.

LeCun, Y., L. Bottou, Y. Bengio and P. Haffner (1998), Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, **vol. 86**, pp. 2278–2324.

Levy, A., G. Konidaris, R. Platt and K. Saenko (2017), Learning multi-level hierarchies with hindsight, *arXiv preprint arXiv:1712.00948*.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra (2019), Continuous control with deep reinforcement learning, *arXiv:1509.02971*.

Nachum, O., S. Gu, H. Lee and S. Levine (2018), Data-efficient hierarchical reinforcement learning, *arXiv preprint arXiv:1805.08296*.

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", in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 136–143, doi:10.1109/ROBOT.2007.363777.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov (2017), Proximal Policy Optimization Algorithms.

Singh, A. (2017), *Anomaly Detection for Temporal Data using Long Short-Term Memory (LSTM)*, Master's thesis, KTH, School of Information and Communication Technology (ICT), Sweden.

Sutton, R. S. and A. G. Barto (2018), *Reinforcement learning: An introduction*, MIT press.

Utanohara, Y. and M. Murase (2019), Influence of flow velocity and temperature on flow accelerated corrosion rate at an elbow pipe, *Nuclear Engineering and Design*, **vol. 342**, pp. 20–28.

Vezhnevets, A. S., S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver and K. Kavukcuoglu (2017), FeUdal Networks for Hierarchical Reinforcement Learning, in *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, Eds. D. Precup and Y. W. Teh, PMLR, International Convention Centre, Sydney, Australia, pp. 3540–3549.
http://proceedings.mlr.press/v70/vezhnevets17a.html

Wang, H., M. Emmerich, M. Preuss and A. Plaat (2019), Hyper-parameter sweep on alphazero general, *arXiv preprint arXiv:1903.08129*.