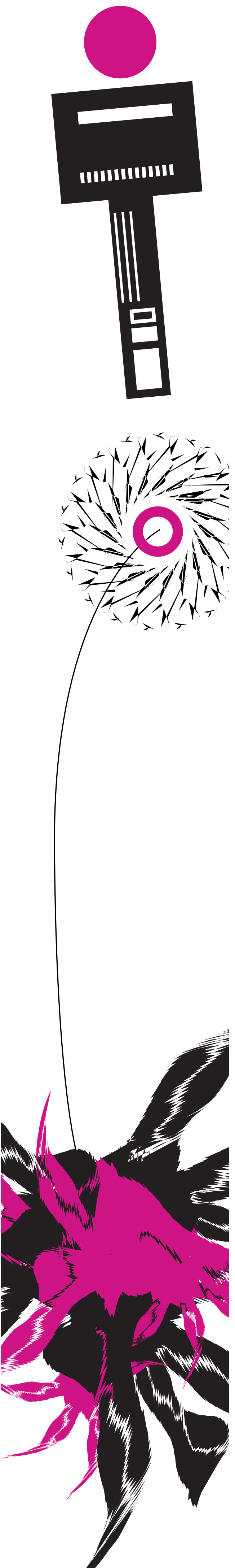

ROBOTICS AND MECHATRONICS

Development and Optimization of a Robot Navigation Algorithm combining Deep Reinforcement Learning and State Representation Learning

R. (Ruben) Obbink

MSC ASSIGNMENT

February, 2020

009RaM2020
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

Abstract

Deep Reinforcement learning is a growing field in robotics. It can help to solve complex tasks that do not have explicit programming solutions. These complex, high-level tasks often require high-dimensional input data such as camera images. One of the challenges that remains using *end-to-end reinforcement learning* is the enormous amount of (experience) data required to solve the task. In robotics, obtaining this data is often time-consuming and costly. State representation learning promises to solve this problem by introducing a network that maps the high-dimensional input data to a low-dimensional state representation before it enters the reinforcement learning algorithm.

In literature, various state representation learning methods are presented. This thesis aims to continue on the work of Jonschkowsk et al. [1], which is a robotics-specific state representation learning method that makes use of the structure imposed by physics to extract relevant task-specific information out of the high-dimensional input data. To contribute to their work, we sought to find more knowledge regarding the optimal state dimension related to the task and investigated the intuitive meaning behind these states. Moreover, we emphasized on more realistic use-cases. This includes transfer learning from simulation to reality and experiments in more realistic environments.

In this work, we make use of a differential drive mobile robot that is tasked to navigate towards a certain target. The mobile robot is equipped with a camera and a 360-degree lidar system. The robot solves the task using a combination of state representation learning and reinforcement learning. The algorithms are implemented in Python using Tensorflow. ROS middleware is used as a communication structure between the high-level controller running in Python and the low-level controller running on the real robot and the simulated system in Gazebo.

Preliminary results indicated that the camera and lidar system can be used complementary, reducing training time by 50% compared to camera input only. Moreover, it was shown that by increasing the consistency of the taken actions, the quality of the state representation can be increased significantly. This resulted in a reduced training duration of 66%. Dedicated experiments regarding the research goal have shown how the optimal state dimension depends on the dimension of the task and the complexity of the mapping problem. Also, it was shown that this dimension can impact training performance significantly. Furthermore, it was shown that the proposed approach was able to achieve great results in a more realistic environment. It was also shown that the learned state representation can be task-general and that it is possible to transfer the state network onto different tasks. Most importantly, it was shown how transfer learning can be employed from simulation to a real robot in order to cut operation costs and reduce training time.

Acknowledgements

At the end of my graduation period, I would like to express my gratitude to the people that helped me during this important period.

First of all, a special thanks goes to my daily supervisor Nicolò Bottehgi for all the time and effort he has put in guiding me, and for all the great discussions we have had. His passion and enthusiasm were really catching and encouraged me greatly throughout the project. Besides his expertise in his field, he is also eager to share his knowledge and ideas. I have had great pleasure in our collaboration, from analyzing results, to performing experiments, to working together on new ideas. It has been a great opportunity to work with him.

Next, I would like to thank my second supervisor dr. Beril Sirmaçek for showing great interest in my work. Her kindness, willingness to help, critical questions and feedback during the meetings have been a great experience throughout the period. Her experience on how to approach things has helped me greatly.

I would like to thank dr.ir Abeje Mersha. As his former student, it has been a great experience to collaborate with him during this project. I would like to thank him for the constructive meetings, for sharing his view in a theoretical but also very practical sense, for the encouraging moments he has given me throughout the project and the genuine interest he has shown. Also, I would like to thank the mechatronics department of Saxion for supplying the setup and for the opportunity to exchange knowledge.

A thanks goes to prof.dr.ir Gijs Krijnen and dr. Mannes Poel for investing their time in me and for taking part in the committee. Also, I want to thank the Robotics and Mechatronics department for the opportunity they gave me and for the limitless supply of coffee.

Without the support of my parents and my girlfriend, I would never have gotten to the point where I am now. I am extremely grateful for the chances they gave me and I hope I made them proud.

Ruben

Contents

1 Introduction

Nowadays robots are used for increasingly complex tasks. Many of these tasks require the robot to operate autonomously and able to make its own decisions. Often, these autonomous tasks do no appear to have an obvious programming solution. A very common example would be that of a mobile robot, moving objects from place to place or performing autonomous inspection. Other examples are learning how to park or drive a car or to control a robotic arm in a confined space.

Reinforcement learning is a field of study that proves itself to be a great solution to these kinds of tasks and has been around for quite some time. More recently, the method has been combined with deep learning. This greatly enhanced the scalability of the method and allowed for solving more complex tasks. Since then, reinforcement learning has even gained greater attention. A recent study has shown that deep reinforcement learning can achieve great results on Atari games [2]. Another example of the capabilities of deep reinforcement learning is AlphaGo [3], which is an algorithm that has learned to play the game Go. Go is considered one of the most complex games for a computer to learn. AlphaGo was first to win from a professional player, using deep reinforcement learning.

More recently, deep reinforcement learning has proven to be a solution to many robotic applications. These applications range from learning a robot to walk [4], to learning a robotic manipulator to open a door [5]. However, applying reinforcement learning directly to complex robotic tasks is often not possible. Reinforcement learning uses so-called rewards to "tell" the robot if an action is either good or bad. This principle is used in order to define the optimal behavior of the robot. The robot learns to relate the inputs, i.e. the observations, and its actions to the reward and thereby learns the optimal policy. Most robotic tasks are complex and require much (multi-modal) sensory input. However, the rewards may not contain much information and are generally indirect of nature. This makes it difficult for a deep reinforcement learning network to make connections between the high dimensional input signal and the uninformative reward signal. This means that the robot needs much experience, i.e. data, in order to fulfill its tasks. In robotics, this is undesirable due to the rather high operational costs.

A fairly recent method to combat this problem is the use of a network placed before the deep reinforcement learning network that is able to compress the input data. This is referred to as (state) representation learning. Jonschkowski et al. [1] have already proven that this method can be very effective in (simple) navigational tasks. Besides that, there are a lot more examples of research that consider state representation learning as a solution to the reinforcement learning dimensionality problem [6, 7, 8, 9]. The idea of state representation learning is mainly to reduce the dimension of the input data before it enters the reinforcement learning network by predicting a low dimensional state from the observations. The advantage of this is that the deep reinforcement learning network does not need to learn this state representation mapping as part of its control policy. This could enhance performance, training duration and generalization.

The aim of the project is to continue on the state-of-art research of the so-called state representation learning. One part of this research aims to investigate the effect of the state dimension on the performance of the reinforcement learning algorithm. This can aid further research by giving insight into the effect of the state dimension and help to choose the right state dimensionality. The other part of the thesis focuses on the applicability of state representation learning in more realistic scenarios and evaluates the feasibility of transfer learning from simulation to a real system in order to decrease learning time and avoid expensive operational costs of robotic equipment.

The use-case adopted in this thesis is a mobile robot that is tasked to navigate from a starting location to a target location as shown in Figure 1.1. The research questions are aimed at this specific use-case. However, the outcome of this research is not exclusively aimed at navigational tasks. The results can be used for any method that implements reinforcement learning using high-dimensional input data.

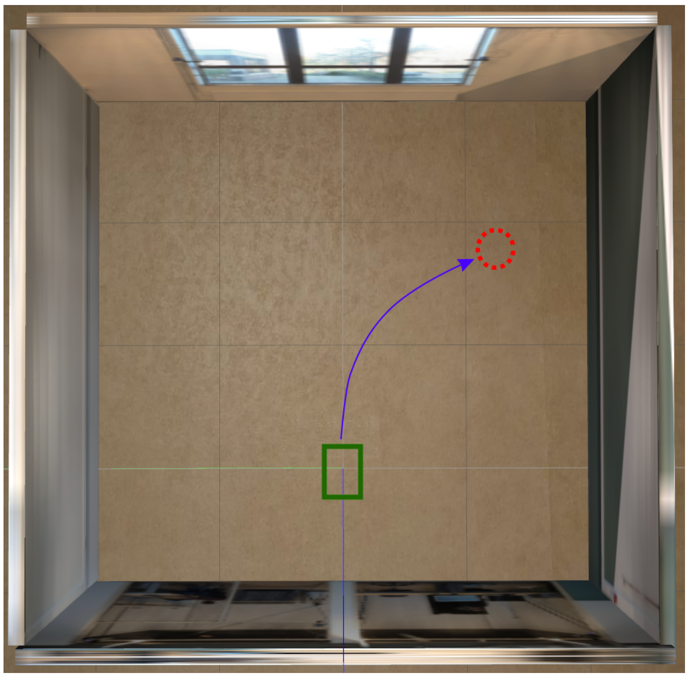

Figure 1.1: Use-case proposal of a mobile robot (green) that is tasked to navigate towards a certain target location (red).

1.1 Problem formulation

The main idea of state representation learning is to yield a state representation that is considerably reduced in dimension compared to the sensory input. This dimensionally reduced state should be able to capture all information that is relevant to the task and should discard all unnecessary information. This more compact form of information allows the reinforcement learning network to converge faster or even allows for a better solution. In most aforementioned state representation learning literature, the chosen state dimension is only slightly discussed upon. However, we believe that the state dimension can play a significant role in performance and/or learning speed. Also, there could be many factors that affect the optimal state dimension. For example, it is possible that a prominent obstacle forces the state representation network to encode information corresponding to the obstacle. Another factor that could affect the optimal state dimension is for example how the reward function is defined. This thesis aims to gain insight into these factors and finally to help find the optimal state dimension in comparable and more complex problems.

Another part of the thesis focuses on real-world scenarios. In literature, relevant research mainly focus on theoretical use cases [1, 8, 6] and gaming environments [7, 9]. The goal is to perform a feasibility study regarding state representation learning in real-world scenarios. Besides that, transfer learning possibilities are investigated. This means that we study the possibility to transfer knowledge obtained trough simulation to a real robot.

1.2 Proposed method

Lesort et al. [10] touch upon several state-of-art state representation learning methods. A distinction is made between several methods, namely observation reconstruction, learning a forward model, learning an inverse model and using prior knowledge. The goal of this project is to continue on a prior-knowledge-based state-of-art state representation learning method [1] with dedicated research questions.

Their research differentiates by developing a loss function that exploits prior (robotic) knowledge to constrain the state representation such that only properties relevant to the task are encoded. Most other methods try to solve a general state representation learning problem in arbitrary (artificial) environments. This means that the state encoding network is based on compressing data as much as possible without information loss rather than extracting data that is actually relevant to the task. This leads to a much more generic state representation at the cost of the compression ratio. According to Jonschkowski et al. [1], using prior (robotic) knowledge is vital to create versatile robots. We share this vision and believe that exploiting this prior knowledge is a very promising approach to the state representation learning problem and very well suited for use-cases that are in line with the research group Robotics and Mechatronics.

For example, this prior knowledge may include the fact that we know that the agent cannot jump from one location to another in just one time step. This knowledge can help form a temporal coherent state representation. Another example is that we expect similar actions to provide a similar result in movement. This can help construct a state representation that distributes the samples spatially according to Euclidean distances between the samples.

In this thesis, we use a fairly basic reinforcement learning method named deep double Q-learning [11]. This is a combination of the well known double Q-learning [12] and the neural network based deep Q-learning [2].

In this research, we make use of a differential drive mobile robot equipped with a lidar and a camera. The camera collects RGB images in front of the robot with a 60-degree field of view. The lidar supplies laser distance data on a 360-degree scan range around the robot, resulting in a cross-sectional view of the environment in the z-direction at a certain height. The total structure of the proposed method is as follows: high dimensional inputs from the lidar system and camera go through the state representation learning network. This network reduces the sensory input dimension, m, to a compressed state representation with dimension n, with $n < m$. The low dimensional state representation then enters the reinforcement learning algorithm.

The reinforcement learning algorithm determines the optimal action from the state prediction. The agent performs this action and obtains experience from the resulting state change. This experience is then fed back into both networks in order to improve future predictions and to develop a better policy. The proposed framework is depicted in Figure 1.2.

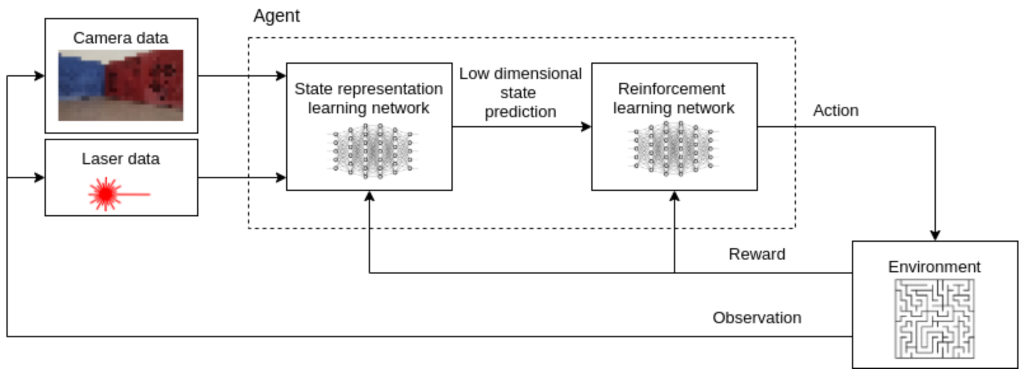
Agent
Camera data
Laser data
State representation learning network
Low dimensional state prediction
Reinforcement learning network
Action
Reward
Environment
Observation

Figure 1.2: Proposed architecture with state representation learning and reinforcement learning.

1.3 Research objectives

Based on the aforementioned goal of the project, two main research questions can be defined. Both questions can be further detailed with sub-questions.

RQ1. What is the optimal output dimension of the state representation learning network for the studied reinforcement learning problem?

RQ1.1. What is the effect of the size of the environment on the optimal state dimension?

RQ1.2 Does the presence of obstacles affect the optimal state dimension?

RQ1.3 How does the reward function affect the optimal state dimension?

RQ2. To what extent can the combination of reinforcement learning and state representation learning be used in real-world scenarios?

RQ2.1. Can the agent be trained to generalize well against multiple target locations?

RQ2.2. Can state representation learning be effective in real-world environments?

RQ2.3. Can transfer learning be used to reduce training duration of an actual robot by pre-training in simulation?

Answering these research questions with dedicated experiments will help to gain insight on the state representation learning method and will help to go from a "black-box" to where we actually understand what happens in between the two networks. Besides that, useful information is gathered regarding the feasibility of state representation learning in real-life scenarios.

1.4 Thesis outline

This thesis presents the implementation of a reinforcement learning algorithm combined with state representation learning as a means to answer the aforementioned research questions. The remainder of this document is structured as follows: **Chapter 2** presents a background study regarding relevant reinforcement learning and state representation learning methods and features a mathematical background on the data visualization methods used in this thesis. **Chapter 3** analyzes the research questions and points out the challenges on the methods that are used. **Chapter 4** features a detailed description on how the experiment environments are set up, features an outline of the software architecture and finally describes the set of experiment designs. **Chapter 5** presents and discusses the results. In **Chapter 6**, a conclusion is drawn and recommendations are given for future work.

2 Background

This chapter presents an overview of the mathematical and conceptual background knowledge that is needed to understand the remainder of this thesis. This chapter gives insight into the reinforcement learning problem and the concept of state representation learning. Also, it will discuss relevant visualization methods.

2.1 Reinforcement learning

Reinforcement learning is one of the three main machine learning paradigms alongside supervised and unsupervised learning. The difference between reinforcement learning and the other two methods is that reinforcement learning does not rely on pre-processed data but rather learns from its own experience [13]. This experience is obtained by interacting with the environment. When the agent performs an action or reaches a specific state, the agent is given a reward. The reward tells the agent if the action was either good or bad. Based on this information the agent is able to learn the desired behavior. This trial-and-error process is very similar to the human learning process and therefore has resemblance to work in psychology [14]. The reinforcement learning model is displayed in Figure 2.1.

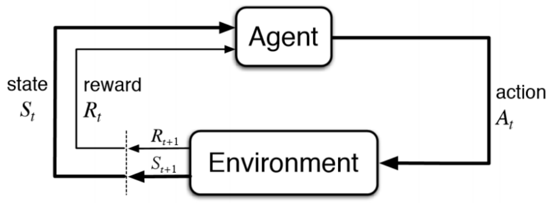

Figure 2.1: Reinforcement learning model [13].

The agent in state S_t performs an action A_t on the environment. At the next time step the agent obtains a new observation, or state, S_{t+1} and a reward R_{t+1}. The agent updates its policy based on reward R_{t+1} and the state action pair (S_t, A_t). This is done iteratively until the algorithm converges. How this update is performed will be further elaborated in this section. The reward model needs to be specified beforehand and should be structured such that the appropriate behavior is achieved. This can be for example to reach a certain position or to discover new parts of the environment. In the beginning, the agent will be extremely incompetent. After several iterations, the agent will start to learn the relation between the inputs and the reward and will start acting towards indented behavior. Training a reinforcement learning algorithm is an iterative process and is generally more time consuming than supervised learning methods since no labeled data is available and the agent needs to learn solely based on its own experience. The rest of this section provides a mathematical background on the reinforcement learning problem and will introduce several important concepts.

2.1.1 Markov property and Markov decision process

In order to understand the Markov decision process [15], we first take a look at the Markov property. The Markov property states the following:

The future is independent of the past given the present.

What this means is that given the current state, the history of states is irrelevant since the state itself gives us the same amount of information about the future as history does. If the state has the Markov property, the decision process becomes much easier because only the current state has to be taken into account.

A state can be said to have the Markov property if and only if the state representation fulfils the above requirement. Mathematically, this can be defined as the following:

$$P\left(s_{t+1}|s_t,a_t\right)=P\left(s_{t+1}|s_1,a_1,\ldots,s_t,a_t\right), \tag{2.1}$$

where P is the probability of ending up in a state s, a is defined as the action that is taken by the agent and t is defined as the discrete time step. The set of actions that can be taken and the states that can be visited by the agent can be either a finite or continuous set. Equation (2.1) basically tells us that the probability of the transition from state s_t to next state s_{t+1} solely depends on the current state s_t and action a_t. Information about the past does not contribute to the state transition.

Next, we consider the definition of a Markov decision process (MDP). A Markov process is most importantly a memory-less process which considers partly random transitions from state to state. A Markov decision process can be represented using four elements:

$$MDP=\left(S,A,P\left(s_{t+1}|s_t,a_t\right),R\left(s_t,a_t\right)\right), \tag{2.2}$$

in which S is a finite set of states that have the Markov property, A is the set of actions that the agent can take, and R is the reward given for the state/action combination.

A MDP is a mathematical framework that can be used to find a policy for the decision-maker. The goal of reinforcement learning is to find a solution to the given MDP that maximizes the reward from the current state onward. This solution is called a policy. The policy decides what action to take being in a certain state.

2.1.2 Model-free learning versus model-based

Reinforcement learning is mainly concerned with obtaining the optimal policy when the MDP is unknown [14]. This leads to two approaches.

A model-based learning method will try to emphasize on constructing an as accurate as possible representation of the MDP using the experience samples. If a reasonably accurate representation is available of the MDP parameters [S, A, R, P], meaning that we know all states, transition probabilities, and the rewards, a planning can be constructed that leads to maximum reward. The problem can then be solved using dynamic programming [16].

Model-free methods assume that the transition probabilities are not known or can not effectively be estimated during training. Therefore, the step of learning the model is skipped and the controller is directly learned based on experience. In robotics, the environment model is often not known and estimating this model would require extensive computational resources [17]. Therefore, model-free methods are often the method of choice in the field of robotics.

2.1.3 Temporal difference

Temporal difference learning [13] is the basis of the so-called Q-learning that will be employed. Temporal difference learning is model-free and therefore does not require the transition probabilities to be known or estimated. Contrary to other methods such as the Monte Carlo method [13], temporal difference methods do not have to wait until an episode is over in order to update the value function. Temporal difference learning uses the so-called bootstrapping, which means that the value function is updated based on estimates. When a new experience sample is obtained, temporal difference learning uses the observed reward in order to update the value function prediction more towards the actual value. The formula for updating the new value is similar to a low-pass filter:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(\overbrace{r_{t+1} + \gamma Q(s_{t+1},a_{t+1})}^{\text{The TD target}} - Q(s_t,a_t)\right), \tag{2.3}$$

in which t is the time sample, $Q(s,a)$ is the action value, α is the learning rate, r is the reward and γ is the discounting factor. The method defines a TD target, which is the observed reward summed to the action value of the new state (given policy π) discounted by γ. The TD error is defined as the difference between the target and the actual prediction of the value. The new value is then updated based on the TD error multiplied by the learning factor. After enough iterations, the value function will eventually converge. The optimal policy is then defined as:

$$\pi^*(s) = \underset{\text{a}}{argmax} Q(s,a). \tag{2.4}$$

Since the update only considers the next state, we can see that temporal difference learning exploits the Markov property. This means that the method is very effective in Markov environments.

Temporal difference learning compared to Monte Carlo based methods suffer a lot more from bias. The reason for this is that the values are initialized randomly in the beginning. These initial values cause the method to be biased. This bias will degrade exponentially in tabular forms of reinforcement learning. However, if temporal difference learning is employed trough deep learning, this can cause stability issues. Although temporal difference learning suffers from bias, Monte Carlo methods tend to suffer more from variance. The temporal difference method only considers the next state, therefore variance is not that much of an issue. However, the return value used in Monte Carlo is based on a lot more experience samples. Therefore, the variance is often a multiple of that of a temporal difference based method.

2.1.4 (Double) Q-learning

Q-learning, proposed by Watkins [18], is a model-free reinforcement learning technique that employs the temporal difference method to update its parameters as described in (2.3). For each combination of state $s \in S$ and each possible action $a \in A$, we store a Q value that tells how good or bad it is for the agent to take action a in state s. The optimal policy is to take the action that corresponds to the maximum Q-value, maximizing the expected return. The algorithm is depicted below:

Algorithm 1: Q-learning.

```
Initialize $Q(s,a)$ randomly, $\forall s \in S$, $\forall a \in A$
for each episode in total episodes do
    Initialize $s_0$
    for each time step $t$ in episode do
        Select action $a_t$ using policy $\pi(s_t)$
        Take action $a_t$ and observe new state $s_{t+1}$ and reward $r_{t+1}$
        Update Q-value using TD error:
          $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right)$
    end
end
```

Q-learning is rather simple to implement and can deliver reasonable performance. However, the method by itself faces some issues regarding over-estimation. This can be explained by the following. Consider an environment with four states: A, B, C and D as displayed in Figure 2.2.

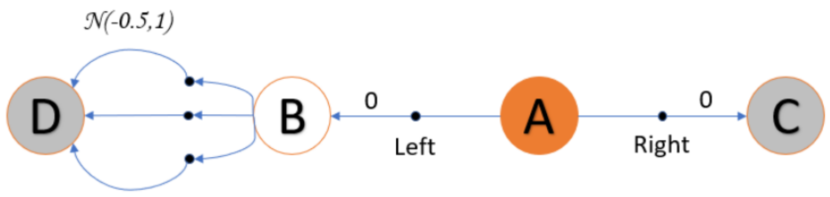

Figure 2.2: Example over over-estimation problem in Q-learning [19].

The agent starts in state A and can choose to go either left or right. If the agent goes to state C, it receives a reward of 0 and terminates. If it goes to state B it will also receive 0 rewards but does not terminate. In state B the agent can choose from a number of actions, but the reward accompanying these actions is randomly selected according to normal distribution $\mathcal{N}$ with a mean of -0.5 and a variance of 1. The agent then proceeds to state D and terminates. When looking at the MDP, it can be seen that taking the right action results in an average return of 0, while the left action will result in an average return of -0.5, meaning that the right action is preferred. However, the Q-learning algorithm might perceive this differently. Because the action value $Q(s,a)$ is based on the maximum of all the Q-values, it is highly possible that one of the actions that can be selected in state B has gained a higher average reward than 0, purely by chance. In state A the action value to go left will be selected according to the reward of state B, which is 0, summed to the maximum of the action values of state B. If one of the actions values happens to be higher than 0, the action value to go left will be higher than 0, meaning that the Q-learning method will over-estimate and choose the wrong action. This phenomenon is called the overestimation problem of Q-learning.

A solution to this problem was proposed by Hasselt et al. [12] by introducing a second independent Q-value table. Every iteration, based on probability 0.5, it is chosen whether the state value of table A or table B is used. If the value comes from table A, the experience sample is used to update table B and if the value is taken from table B, it is used to update table A. In this way, an unbiased prediction can be made for the action value. Several years later, Hasselt introduced an updated version of the double Q-learning algorithm [11]. This updated version was designed for deep Q learning, which will be later discussed in this chapter. This method employs a model Q and a target Q'. In which Q' slowly or periodically takes over the values of Q. We use Q to select the action and we use Q' to determine the predicted action value. The TD target (indicated in (2.3)) is here defined as Q^* and can be calculated as:

$$Q^*\left(s_t, a_t\right) \approx r_{t+1} + \gamma Q'\left(s_{t+1}, \underset{a_{t+1}}{\operatorname{argmax}} Q\left(s_{t+1}, a_{t+1}\right)\right) \tag{2.5}$$

According to Hasselt et al. [11], the over-estimation can be reduced by decomposing the max operation into the action selection and the action evaluation. This claim is backed up by results that show a significant improvement in Atari games compared to Mnih et al. [20]. An important feature of the new algorithm is that it is very similar to standard deep Q-learning. The network structure can be kept equal. The only thing that needs to be changed is that a target network should be defined that slowly or periodically takes over the values of the online network and that the TD target is now calculated using (2.5). The updated algorithm is depicted below.

Algorithm 2: Double Q-learning.

```
Initialize $Q(s,a), Q'(s,a)$ randomly, $\forall s \in S$, $\forall a \in A$
for each episode in total episodes do
    Initialize $s_0$
    for each time step $t$ in episode do
        Select action $a_t$ using policy $\pi(s_t)$ according to $Q(s_t, a_t)$
        Take action $a_t$ and observe new state $s_{t+1}$ and reward $r_{t+1}$
        Update Q-value
            $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma Q'\left(s_{t+1}, \underset{a_{t+1}}{\text{argmax}} Q(s_{t+1},a_{t+1})\right) - Q(s_t,a_t)\right)$
    end
    if target network update then
        Update target network
            $Q' \leftarrow Q$
    end
end
```

Note that the algorithm above is based on a tabular form of Q-learning for comparison purposes. In Hasselt et al. [11], the method was proposed for deep Q-learning which will be further elaborated below.

2.1.5 Deep reinforcement learning

Deep reinforcement learning is a combination of reinforcement learning and deep learning. The use of neural networks in reinforcement learning greatly increases its scalability. In standard reinforcement learning, each state-action-pair value needs to be stored in memory. When the set of states and the set of actions becomes larger, storing the values in tabular form will create memory and performance issues. To counteract this problem, neural networks can be used as function approximators in order to map the state-action pairs to the rewards. Moreover, the neural-network-based variant also has an advantage in continuous state or action spaces: no discretization has to take place to be able to store them in a table. The network receives a state and will predict a Q-value for each possible action. The difference between the tabular form and the neural-network-based form is visualized in Figure 2.3.

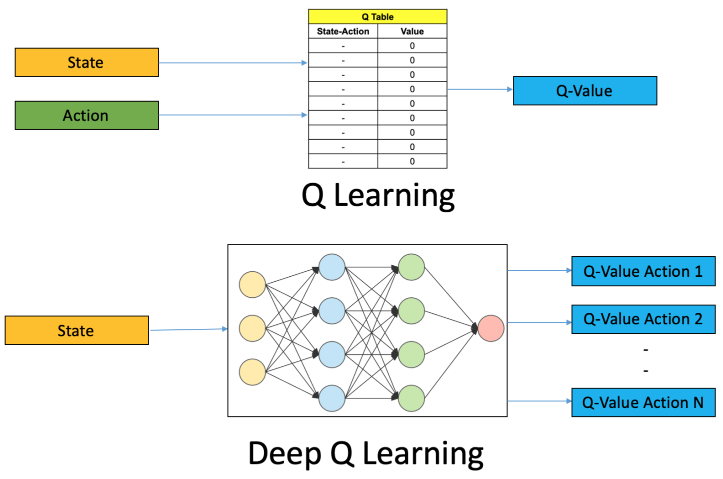

Figure 2.3: Deep reinforcement learning versus tabular form [21].

Another advantage of neural networks is that it enables the use of convolutional networks. Convolutional networks allow for extracting certain features from high dimensional input data. The use of convolutional networks can highly improve the performance of reinforcement learning methods [2].

2.1.6 Experience replay

Neural networks generally perform better if there is no temporal correlation between the samples [22]. In other words, the samples are independent and identically distributed. If the neural network sees too many samples of one kind, it tends to forget the others. In reinforcement learning, samples are gathered from timestamp to timestamp, meaning that this can cause problems. Fortunately, there is a solution to this kind of problem that has proven to be very successful. This method is called experience replay [23] and has become of high importance in deep reinforcement learning. Recently, promising results have been shown by Schaul et al. [24] on Atari games using Google's DeepMind, where they used an adaptation that prioritizes samples based on the significance of the state transition.

In contrast to just using the experience sample of the last time step, experience replay uses one or more randomly selected experience samples from its memory. This breaks the time-based relation between the samples and mitigates the instability issues that deep reinforcement learning has. The memory, or replay buffer, has a finite size with a specified amount of samples. The amount of samples does have an effect on the learning speed and does have to be chosen properly. Alternatively, a method can be used that dynamically adjusts this size [25].

Another advantage is that the experience is used more efficiently since the samples are used multiple amounts of times. In robotic applications, experience is sparse and therefore efficient use is highly encouraged. Because the Q-values converge very slowly, taking multiple passes of acquired data is really beneficial.

2.1.7 Exploration versus exploitation

Another interesting subject regarding reinforcement learning is the exploration-exploitation problem. In the beginning, the agent knows nothing about the environment and takes more or less random actions. After some iterations, a reasonable solution is found. The problem is that if the robot always would follow the current optimal path, the robot can miss certain parts of the environment that could yield even greater return. However, if the robot always tries to explore the environment, the agent does not learn to reach its goal. This is called the exploration-exploitation dilemma.

There are several methods that deal with this problem, a well-known method is the ϵ-greedy policy. The method is very simple: based on probability the robot will either take a greedy, most optimal, action or take a random action that could provide the agent with new information. The chance that one of the the two actions is selected is based on the parameter ϵ. This parameter can be changed online and is usually set to favor random actions during the beginning of the training and to follow the optimal policy at a later stage.

2.2 State representation learning

The term state refers to different concepts in control theory and reinforcement learning. In control theory, the state of an agent often is referred to as the position, angle and their respective velocities and accelerations (for navigational problems). In reinforcement learning frameworks, the notion of the word state is quite different. In reinforcement learning problems, the state corresponds to the specific configuration of input data. Often, this input data is coming from sensors and therefore not directly linked to the states used in control theory. This means that for example the state of the agent can be represented by a specific configuration of pixels from a camera or a set of distance values obtained by a laser distance sensor.

The use of a camera or a lidar system comes at the cost of having high-dimensional input data. Most of the time, this is unavoidable. A low dimensional fully informative ground truth state, i.e. GPS-data, is not always available. Sensors such as a camera or lidar are used as a replacement (drift-free) measurement system. This means that the reinforcement learning algorithm has to deal with high dimensional input data.

The high dimensional input data carries much task-irrelevant data. This, together with the relatively uninformative rewards makes it very challenging for the reinforcement learning algorithm to learn to the mapping that leads to the highest reward [7]. This means that a high number of experience samples are required. In robotics, this is often undesirable due to high operational costs. Reducing this input data to a more compact form, losing as small information as possible relevant to the task can be considered as the work field of feature engineers. In this case, the mapping from observation to a more compressed state representation $f : O \to S$ is done by hand. Feature engineering allows robotics to solve more complex tasks. However, the observation-to-state-mapping depends on the task, meaning that for each task this mapping has to be redefined.

State representation learning uses machine learning to extract relevant information from the high dimensional inputs. State representation learning is described as the process of learning the mapping from observations to a state representation [1]. A very nice property of state representation learning is that multiple types of sensors can be used as an input. Without specifying the importance or accuracy of the sensor, the method will figure out which sensor is more important and which is less.

Since the optimal state representation is not known beforehand, state representation learning can be considered as an unsupervised learning problem. The current state is not available to measure the fitting error and therefore another sort of loss function should be defined. In Caselles-Dupré et al. [8], the encoding property of a variational autoencoder [26] is used to compress the state dimension. The loss function is based on the difference between the initial image and the reconstructed image. This ensures that the encoded reduced dimension state does not lose information compared to the input data. The encoded part is then fed to the reinforcement learning algorithm. Overall, the research shows a significant increase in performance of the reinforcement learning algorithm using the learned state presentation compared to the raw data input.

Jonschkowski et al. [1] uses so-called robotic priors as a loss function. These priors are assumptions about the world around us. An example of these priors is the repeatability prior: same actions from the same state should produce similar changes in states, or the temporal coherence prior: states should gradually change over time. One could understand that a combination of such priors can describe the world in a competent manner and could be used to learn a mapping from observation to a dimensionally reduced state. The results of that research show a significant increase in performance as well as generalization when using the learned state representation on a fitted Q-iteration reinforcement learning method compared to the bare reinforcement learning method.

Yet another research makes use of these priors. Munk et al. [6] uses so-called predictive priors. The idea is to create a model network that is able to predict the next state and the next reward from the current observation and the taken action, as being shown in Figure 2.4. This means that the unsupervised learning method is reduced to a supervised learning method after the action has been performed. After this model network is trained, this mapping is partly incorporated into an actor-critic reinforcement learning algorithm by copying the part that maps the current observation to the current state as can be seen in Figure 2.5. After that, the reinforcement learning algorithm can be trained. Results show an increase in performance using the model learning variant in comparison with a standard deep deterministic policy gradient (actor-critic) by testing on a 2-link robot arm benchmark and another well-known benchmark that consists of controlling an octopus.

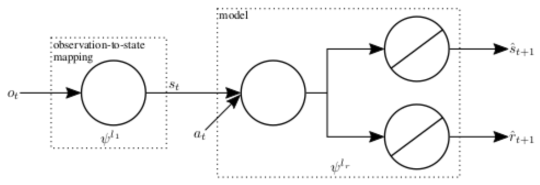

Figure 2.4: Model network incorporating the predictive priors [6].

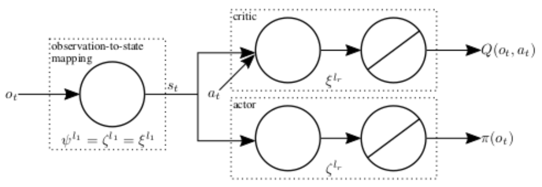

Figure 2.5: Actor critic network with integrated observation to state mapping [6].

A more recent paper of partly the same authors compares different types of state representation learning [7]. Bruin et al. compare several methods (including above-mentioned researches) such as: using an autoencoder, using a reward prediction method, using the slowness and diversity property and learning the (inverse) dynamics. These methods are more explicitly described in the papers. The results have shown that the slowness and diversity and the inverse dynamics have a considerable impact on the generalization of the robot, while the autoencoder proved to have faster learning speed and stability. Also, they claim that training the state representation learning and reinforcement learning network simultaneously, allows the reinforcement learning process to help shape a better state representation. This overall shows a minor increase in performance in training but shows enormous improvements during testing.

2.3 Data visualization

In this thesis, high dimensional data will be analyzed. In order to still be able to visualize this data, we make use of two methods that can reduce the dimension of the data set while retaining the initial data structure as much as possible.

2.3.1 Principal component analysis

As the dimension of the data grows, so does the difficulty to visualize it. In order to still be able to visualize the data, we can reduce the dimension to a more comprehensible size. One of the methods for doing so is called principal component analysis. Often, data sets have many correlated variables, meaning that some variables only add partial or even no information and therefore can be removed without data loss.

Principal component analysis can help reduce the dimension of the data set such that n dimensions with high covariance (linearly dependent) are replaced by a single dimension that is a linear combination of those n dimensions. By doing so, principal component analysis converts a data set with possibly linearly correlated variables or dimensions into a set of uncorrelated variables with equal or fewer dimensions.

In order to obtain the mapping, first we need the covariance matrix:

$$\mathbf{S} = \begin{pmatrix} s_{11} & s_{12} & s_{13} & \cdots & s_{1k} \\ s_{21} & s_{22} & s_{23} & \cdots & s_{2k} \\ s_{31} & s_{32} & s_{33} & \cdots & s_{3k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{j1} & s_{j2} & s_{j3} & \cdots & s_{jk} \end{pmatrix}. \tag{2.6}$$

The indexes of the covariance matrix are symmetric and can be calculated using formula (2.7).

$$s_{jk} = \frac{1}{n-1}(X-\bar{X})(Y-\bar{Y})^T, \tag{2.7}$$

with X being the data set that corresponds to dimension j and Y being the data set that correspond to dimension k and $\bar{X}$, $\bar{Y}$ being their mean.

The non-diagonal terms of the matrix describe the covariance between two variables and the diagonal terms describes the variance for a single variable. As described above, the goal of principal component analysis is to realign the data such that diagonal terms (variance) is maximized and the non-diagonal terms become zero. The principal components can be found by solving the eigenvalue problem [27]:

$$S\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i \quad i = \{1,2\ldots M\}, \tag{2.8}$$

where u_i is the eigenvector corresponding to eigenvalue λ_i and M is the number of dimensions in the initial data set. The largest eigenvalue corresponds to the eigenvector (direction) that explains the most variance, this eigenvector is called the first principal component. The rest of the principal components can be defined in an incremental fashion by selecting the next direction that maximizes the projected variance. These principal components are orthogonal to the principal subspace and therefore uncorrelated.

The data set can then be analyzed by projecting the initial data set onto the principal components using the dot product.

$$\mathbf{p}_i = X \cdot \mathbf{u}_i. \tag{2.9}$$

The projected data p is then reduced to a required dimension based on the number of principal components taken into account. Based on the amount of variance explained by a certain component, we can get an idea of the amount of data loss that occurs when projecting the data onto a lower dimension. The variance explained by a component is often expressed in terms of a percentage/ratio and can be calculated by normalizing the eigenvalues.

$$V_{explained} = \frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i}. \tag{2.10}$$

As shown in this section, principal component analysis is a linear mapping (projection). This means it can only capture linear structures in the variables. However, since the networks are not restricted to a linear mapping, principal component analysis can be limiting the ability to clearly see clustering and/or disentanglement in the learned state representation.

2.3.2 T-SNE

T-Distributed Stochastic Neighbor Embedding (T-SNE) [28] can be used to visualize high-dimensional data. It is able to take high-dimensional data and map it to a lower-dimensional space while retaining most information. Unlike PCA, T-SNE is not a linear projection. By making use of local relationships between points, non-linear clustering properties can be represented as well. The first step of T-SNE is to create a probability distribution over neighboring points. The similarity between two data points x_i and x_j is defined as the conditional probability that point x_i would pick point x_j as its neighbor if neighbors were selected proportionally to the probability density of a Gaussian centered at point x_i. For close points, probability $p_{j|i}$

would be relatively large. For large points $p_{j|i}$ is almost zero. The conditional probability of $p_{j|i}$ is given by

$$p_{j|i} = \frac{\exp\left(-\left\|x_i - x_j\right\|^2 / 2\sigma_i^2\right)}{\sum_{k\neq i} \exp\left(-\|x_i - x_k\|^2 / 2\sigma_i^2\right)}, \tag{2.11}$$

in which σ_i is the variance of the Gaussian distribution placed on point x_i. In T-SNE the number of neighbors should be roughly the same for all samples. This makes sure the influence of each point is equalized. This can be achieved by scaling σ according to the density of an area. T-SNE identifies a hyperparameter called the perplexity that determines the (average) amount of neighbors for any point. The perplexity determines whether we look at a local or a more global structure.

The joint probability of two points is defined as:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}, \tag{2.12}$$

with n as the dimension of the data set. As explained in Maaten et al. [28], this circumvents problems with outliers and ensures that all samples contribute to the cost function. This ensures all regions are well defined.

The goal of T-SNE is to recreate a low dimensional space that is able to mimic the probability distribution of the neighboring points in the higher dimension as well as possible. This is done by enforcing the same distribution in the low dimensional space. A common problem by mapping from high to lower dimensions is the crowding problem, meaning that samples are placed very close in the low dimensional space. T-SNE solves this problem by using a t-distribution rather than a Gaussian distribution in the low dimensional space. This distribution has a much slower drop-off in probabilities for values far from the mean. This means that moderate distances in the higher dimensional space will be spaced further away, preventing crowding of points in the lower dimensional space while still retaining a clustering of points that are very close. Using the Student's t-distribution, the probabilities of the samples q_{ij} of the low dimensional space can be defined as

$$q_{ij} = \frac{\left(1 + \left\|y_i - y_i\right\|^2\right)^{-1}}{\sum_{k\neq 1}\left(1 + \left\|y_k - y_l\right\|^2\right)^{-1}}, \tag{2.13}$$

where y_i and y_j are the new data points of the embedded samples. The embeddings are then optimized using gradient descent by minimizing the Kullback-Leibler divergence:

$$KL(P\|Q) = \sum_{i\neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}. \tag{2.14}$$

The solution is found iteratively, rearranging the points such that they reduce the loss. This eventually results in a map that reflects the local relationships between the high-dimensional inputs as well as possible. Since the loss function is non-convex, there is a risk of getting stuck in local minima.

2.4 Summary

This chapter provided most concepts needed in order to understand the remainder of this thesis. First, it was explained how Q-learning can be applied to a reinforcement learning problem.

It was also discussed how the problem of overestimation in Q-learning can be tackled by keeping an extra network next to the original that slowly takes over the values of the other network. Then it was discussed how neural networks can be used together with reinforcement learning to enhance scalability and work in continuous state and action spaces, resulting in a final algorithm called deep double Q-learning. Deep double Q-learning was chosen because it is easy to implement and can provide reasonable performance. Next, a basic introduction to state representation learning was provided and it was discussed how several different kinds of research apply this idea. Since the method can be considered as an unsupervised learning method, another sort of loss should be used. For example, this can be done using an autoencoder, using a reward prediction method, or using prior knowledge about the task. Finally, we have discussed how we will analyze our high dimensional data by making use of two methods that can reduce the dimension of the initial data set. One of the methods uses linear projections on the dimensions that explain the most variance (PCA). The other is a non-linear method that tries to retain the structure of the initial data set as much as possible by making use of local relationships between points (T-SNE).

3 Analysis

The main goal of this thesis is to present a reinforcement learning solution that is able to navigate a mobile robot from an initial position to a certain target location by implementing and optimizing a state representation learning method. As aforementioned, this thesis focuses on two branches mainly: an investigation of the optimal state dimension and a feasibility study towards real-world applications. This chapter mainly focuses on analyzing appropriate solutions and distinguishing parameters that are considered important to the above-mentioned project goal and research objectives.

3.1 Research objectives

In the first chapter, two main research questions were defined along with corresponding sub-questions. In this section, we analyze these questions and discuss the reasoning behind them.

Let us start with **RQ1** *"What is the optimal output dimension of the state representation learning network for the studied reinforcement learning problem?"* As described in the first chapter most of the state-of-art research do not emphasize on the state dimension. Jonschkowski et al. [1] do actually comment on the state dimension. However, most other researches merely mention the selected size but do not really justify this choice. We think that the state dimension can play a significant role in terms of performance and learning speed. For example, a significantly higher than needed state dimension is most likely able to capture all relevant information of the environment, but does not solve the curse of dimensionality [16]. With a state dimension that is too small, not all relevant information can be captured, which could harm performance. These possibilities are important to consider when choosing the right state dimension. Therefore, we think it is important to further investigate the optimal state dimension. Moreover, we think there are several factors that affect the optimal size of the state dimension on which we based several sub-questions:

The first of these questions is **RQ1.1** *"What is the effect of the size of the environment on the optimal state dimension?"* For a larger environment, the amount of samples that need to be encoded is obviously larger. However, if the structure of the environment remains the same, the difficulty of the mapping problem does not necessarily have to change.

Next, we consider **RQ1.2** *" Does the presence of obstacles affect the optimal state dimension?* We might find that the state representation learning algorithm needs more states to correctly represent the mapping from inputs to a fully informative state representation since an obstacle causes quite a disruption in the reward space.

The final sub-question that corresponds to RQ1 is **RQ1.3** *"How does the reward function affect the optimal state dimension?"* Since the state representation is predominantly built on the reward space, we can expect to see different behavior when a different type of reward function is defined.

RQ2 *"To what extent can the combination of reinforcement learning and state representation learning be used in real-world scenarios?"* This question arises when looking at most of the state-of-art researches. Most of the state representation learning methods only focus on simulation or highly simplified environments. In our opinion, it would be very beneficial to investigate the method's applicability in real-world scenarios. This question is very broad and is therefore divided into several sub-questions.

First, we consider **RQ2.1** *"Can the agent be trained to generalize well against multiple target locations?"* Most of the real-world scenarios require the agent to be flexible and to be able to reach more than one target location. Therefore, it is important to know if and how the state representation can be trained to generalize well enough to allow the reinforcement learning method to reach multiple locations. In this research, we emphasize on the generalization properties of the state representation learning method.

The next sub-question is **RQ2.2** *"Can state representation learning be effective in real-world environments?"* Real-world environments are generally less attractive to test on because they are less controllable and there are often a lot more noise and distractions present. However, in our case, the state representation learning method uses convolutional layers that can make use of the abundance of features that are normally present in more realistic environments. Therefore it would not be surprising that the agent can benefit from these.

Lastly, we have **RQ2.3** *"Can transfer learning be used to reduce the training duration of an actual robot by pre-training in simulation?"* The problem of the high operational cost of robotic systems has already been addressed multiple times. For reinforcement learning in robotics, we need to either be really efficient when training or we should move the training part to simulation instead. This is called transfer learning. Bruin et al. [7] show how pre-training can be used to speed up the training process from simulation to simulation. In this case, we investigate the possibility to do pre-training in simulation and to perform evaluation on the real robot.

3.2 System analysis

In this section, we analyze the main components of the total system. The main components for this system are the reinforcement learning algorithm, the state representation network, and the mobile robot.

3.2.1 State representation learning method

The goal of state representation learning is to find a mapping $\hat{\phi}$ between a high dimensional observation o_t to a lower dimension state prediction $\hat{s}_t$ by discarding all task irrelevant information while keeping the important information. The mapping can be described as $\hat{s}_t = \hat{\phi}(o_t)$.

In Section 2.2, several state-of-art state representation learning methods were presented. During this thesis, we continue using the robotic priors [1]. The research shows a use case similar to the one presented in this thesis and is aimed at real-world scenarios. The method uses five so-called robotic priors, which are assumptions we can make about interacting with the physical world. We know that an agent in the real world should follow the laws of physics. By making use of this knowledge, the learning process can be accelerated.

Robotic priors

In this paragraph, the priors presented in Jonschkowski et al. [1] are discussed. The priors are based on the laws of physics and have been converted into an optimization problem by translating them into a loss function.

The first of the five priors is the **simplicity prior**. *"For a given task, only a small number of world properties are relevant."* [1] This prior assumes that the very high dimensional input data contains a lot of task-irrelevant information, meaning that we can significantly reduce the state dimension without losing information that is important for the given task. This prior has not been converted into a loss function but is automatically introduced when fixing the dimension of the state to a lower number than the input data dimension.

Next we have the **temporal Coherence Prior**. *"Task-relevant properties of the world change gradually over time."* [1] The robotic priors have been largely inspired by Newtonian mechanics. The temporal coherence prior can be compared to Newton's first law of motion: Objects have mass or inertia and their velocity only changes gradually over time as a result of external forces. This means that the difference between two consecutive states should be gradual. The agent cannot be in one part of the room and be at the other side of the room at the next time sample (considering the forces are moderate and the time samples are small). This can be easily translated into a loss function by penalizing Euclidean distance between two consecutive states (3.1). By taking the square we can emphasize large differences or outliers.

$$L_{\text{temp.}}(D,\hat{\phi}) = \mathbf{E}\left[\|\Delta\hat{s}_t\|^2\right], \tag{3.1}$$

where D are the experience samples $D = \{o_t, a_t, r_t\}$, the state change is denoted as $\Delta\hat{s}_t = \hat{s}_{t+1} - \hat{s}_t$ and $\hat{\phi}$ is the observation to state mapping. This loss function keeps consecutive states close and pulls the total state representation together.

Next, we have the **proportionality prior**. *"The amount of change in task-relevant properties resulting from an action is proportional to the magnitude of the action"* [1] This prior is shaped according to Newton's second law of motion: The acceleration of an object is proportional to the resultant of force and its mass. The magnitude of the action therefore also determines the change in state. This prior enforces a state representation that experiences a state change that is proportional to the magnitude of an action. In our case, we do not apply a force but generate a setpoint velocity for a low-level controller. However, this prior does still apply. When a low magnitude action (low velocity command) is applied we expect a small state change. If we apply a higher magnitude action (high velocity command) we expect a more significant state change. This prior enforces that the distances of the states are more or less equalized and that the states are spaced according to their Euclidean distance. The loss function can be realized by selecting experience samples in memory where equal actions are performed and by minimizing their difference in state change. Again, the squared difference is taken to penalize larger differences more heavily.

$$L_{\text{prop.}}(D,\hat{\phi}) = \mathbf{E}\left[\left(\|\Delta\hat{s}_{t_2}\| - \|\Delta\hat{s}_{t_1}\|\right)^2 | a_{t_1} = a_{t_2}\right]. \tag{3.2}$$

Next we consider the **causality prior**. *"The task-relevant properties, together with the action, determine the reward."* [1] The easiest interpretation of this prior is that if we have two equal actions leading to two different rewards, we know that these states must be different and therefore driven apart. For the loss function, we first gather experience points that share the same action but result in different rewards. Then we define a penalty that is an inverse exponential of the squared Euclidean distance. This means that two consecutive states with equal actions and different rewards are penalized heavily for being close together. Therefore, states that lead to different rewards when applying the same actions are pushed apart.

$$L_{\text{caus.}}(D,\hat{\phi}) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2} | a_{t_1} = a_{t_2}, r_{t_1+1} \neq r_{t_2+1}\right]. \tag{3.3}$$

Finally, we have the **repeatability prior**. *"The task-relevant properties and the action together determine the resulting change in these properties."* [1] If the same action is applied in a state that is similar or close in distance, we expect a similar state change. If the same action is applied and the state change is different, we know that these states should be far apart. This enforces a deterministic state representation. The loss function is defined by the squared difference between the state change of similar action samples, scaled by the distance between the states. This means that states with equal actions and different state changes are penalized heavily if they are close, and therefore are driven apart.

$$L_{\text{rep.}}(D,\hat{\phi}) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2} \|\Delta\hat{s}_{t_2} - \Delta\hat{s}_{t_1}\|^2 | a_{t_1} = a_{t_2}\right]. \tag{3.4}$$

The total loss function is defined as the sum of all individual losses. Each individual loss can be weighted by ω. Using these weights we can define the importance of each prior.

$$\begin{aligned} L(D,\hat{\phi}) &= \omega_{\mathrm{t}} L_{\text{temporal coherence}}(D,\hat{\phi}) + \omega_{\mathrm{p}} L_{\text{proportionality}}(D,\hat{\phi}) \\ &+ \omega_{\mathrm{c}} L_{\text{causality}}(D,\hat{\phi}) + \omega_{\mathrm{r}} L_{\text{repeatability}}(D,\hat{\phi}) \end{aligned} . \tag{3.5}$$

From the presented priors it becomes obvious that this state representation learning method relies on an agent that performs actions and gathers rewards. Therefore, it does not work on passive systems that only gather observation samples. However, by using the rewards and actions, the learned state representation encodes only the information that is relevant to the task. Methods that are only based on observations do not know the task and therefore a much more general state representation is shaped.

Adaptation to these priors

Next, we discuss a new approach to these priors [29].

The first prior that has been altered is the **proportionality prior**. In the previous section, the selection criteria of the proportionality prior was based on equal actions. The new selection criteria is based on reward variation. This means that we now select pairs that have similar or equal variation in reward. By doing this, the network shapes a state representation such that pairs with similar reward variation have a similar magnitude in state variation. The new prior can be described as follows:

$$L_{\text{prop.}}(D,\hat{\phi}) = \mathbf{E}\left[\left(\|\Delta\hat{s}_{t_2}\| - \|\Delta\hat{s}_{t_1}\|\right)^2 | \Delta r_{t_2} \sim \Delta r_{t_1}\right], \tag{3.6}$$

where $\Delta r_t = r_{t+1} - r_t$.

The second adapted prior is the **repeatability prior**. In a similar fashion to the proportionality prior, the repeatability is now too selected on equal reward variations rather than equal actions. The purpose of this prior is to push states with similar reward variations but dissimilar state variation apart since these states are different and therefore should not be close.

$$L_{\text{rep.}}(D,\hat{\phi}) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2} \|\Delta\hat{s}_{t_2} - \Delta\hat{s}_{t_1}\|^2 | \Delta r_{t_2} \sim \Delta r_{t_1}\right]. \tag{3.7}$$

The idea of changing the two priors from action-based selection to reward-based selection is that we believe that the reward space of an environment includes a significant amount of information that can be particularly helpful for building a competent state representation. Also, this effectively separates the actions from the state representation learning method. This allows for an easy transition between reinforcement learning methods. For example, the reinforcement learning method can be easily changed to produce continuous actions rather than discrete actions without the need to alter the state representation learning method.

During this thesis, these two new priors are used to obtain the results. However, the scope of this thesis does not lie within the optimization and/or analysis of these new priors as this is part of a research parallel to this.

State representation learning parameter analysis

Several parameters that concern the state representation learning method are analyzed below:

- *Learning rate (α)*: Determines how fast the weights are updated and therefore how fast the observation to state mapping is altered.
- *Regularization rate (λ)*: Determines how much large weights are punished. It helps to combat overfitting.

- *Batch size*: Number of samples that are used to update the state representation learning network each update step.
- *Priors weighting (ω)*: Determines how the priors should be weighted. A configuration that scales all priors equally is favored.
- *Step interval*: Interval in which steps are performed. In each step, the reinforcement learning method performs an action and gathers a new experience sample. For state representation learning, this step time is very important since many priors are based on the difference between samples. This means that the step size highly affects how the state representation is formed. If the step size is too large, the difference between the observation samples becomes too large to be able to form a coherent state representation.

3.2.2 Reinforcement learning method

As described in Chapter 1, deep double Q-learning is adopted as the reinforcement learning solution similar to Bruin et al. [7]. The main reason behind this choice is that the method is easy to implement, provides reasonable performance and is similar to the method used in the priors research [1], who implemented a fitted Q-iteration based on normalized radial basis function. The reason behind choosing a more complex function estimator is that we also test on more complex environments. The reason for upgrading to double Q-learning is that it is easy to implement and it tackles the problem of overestimation. There exist more complex and possibly better-performing reinforcement learning methods. Although, since the scope is mainly on the state representation method we do not look further into these. Deep double Q-learning provides a robust and intuitive basis to assess the state representation learning method.

First, we consider a choice that is relevant in any implementation of reinforcement learning combined with state representation learning. The state representation learning and reinforcement learning rely on each other to get to the final solution. The reinforcement learning method cannot find a good solution without a good representation and the state representation network cannot learn without some experience gathered by applying the reinforcement learning algorithm. Therefore, it is important to consider how these methods work alongside each other and how the networks are updated. There are more or less two solutions.

- Reinforcement learning method is updated every iteration and the state representation is updated every n episodes.
- After n episodes both the reinforcement learning method and the state representation learning method are updated based on the gained experience.

In this research we have chosen for the first option because of mainly the following reason: when the reinforcement learning method is directly updated after each episode, it adapts its policy immediately and therefore acquires more relevant experience samples. This benefits both the reinforcement learning and the state representation learning method. The number of steps in-between these updates is very important for learning because the reinforcement learning algorithm needs enough time to adapt to the new state representation.

Next, we consider an important decision for facilitating the learning process in reinforcement learning specifically for the navigation problem:

- Perform (semi) random exploration and program manual obstacle avoidance and run for a set number of time steps.
- Reset the robot after hitting an obstacle and start from the initial position.

- A combination of these two by implementing obstacle detection, and to reset the agent whenever the target is reached or after a (short) number of episodes.

At first glance, the obstacle avoidance method seems more convenient since it does not require a reset which is inconvenient on a real robot. The problem with this is that when the environment grows considerably larger, the time it takes the robot to capture the whole environment is much longer. A method that only explores relevant regions is prioritized since it speeds up the learning process. Therefore, this is the method of choice in most presented environments.

However, in certain cases is it desirable to have obstacle detection. In more complex environments, obstacle detection can help to reach harder parts of the environment by reducing the number of crashes close to the starting point. In these environments, the third method is employed.

Reinforcement learning parameter analysis

Let us now consider other tuning parameters of the reinforcement learning network. Below, parameters are listed that are considered important during experiments.

- *Learning rate (α)*: Determines how fast the weights are updated and therefore how fast the policy changes after a new experience sample. A too high learning rate can cause convergence problems.
- *Exploration rate (ϵ)*: Determines the trade-off between exploration and following the optimal policy as described in 2.1.7.
- *Regularization rate (λ)*: Determines how much large weights are punished. It helps to combat overfitting.
- *Experience sample memory size*: Determines how many samples are stored in the experience replay batch to train from. Choosing the right size is important since a too small memory causes the temporal correlation to take effect. Taking a list that is too large means that you are training on old state representation samples that are not valid anymore.
- *Batch size*: Since we employ experience replay (2.1.6) we can use more than the current experience sample to update the RL network. The batch size determines how many samples this should be.
- *Discount rate (γ)*: Determines how much rewards in the present are favored over samples in the near future.
- *Reward function*: Inherently defines the behavior of the agent. The agent will eventually follow the path that leads to the maximum reward. By shaping the reward function in a smart way the learning process can be accelerated.
- *Movement characteristics*: The forward velocity and rotational velocity can highly affect performance. The faster the robot is, the harder it is to avoid obstacles. Moreover, if either the rotational or transversal velocity is too high, one action might already overshoot the desired direction of the robot. This is also affected by the frequency in which controls are performed, determined by the step interval depicted in 3.2.1.

3.2.3 Turtlebot analysis

The Turtlebot [30] is used as an agent for both simulation and real-world experiments. More specifically, the Turtlebot 3 is used (Figure 3.1). This is the newest version of the Turtlebot and includes a rotating distance sensor (lidar) and a camera as can be seen from the figure.

The main reason for choosing the Turtlebot is that it is easily available, features all necessary equipment and is easy to implement in both simulation and real-life since a significant amount of open-source code is available.

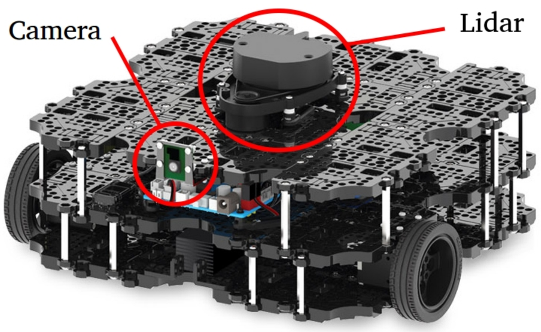

Figure 3.1: Turtlebot 3 "waffle" version [30].

Input data

As aforementioned, the input configuration has been chosen as an RGB camera together with a lidar system that is able to supply 360-degree distance data around the robot. Jonschkowski et al. [1] uses only a camera. However, this camera has a field of view of 300 degrees. Since we use the turtlebot (that is equipped with a lower field of view camera), we use the lidar system as an extension of the camera. This can provide more information on the surroundings of the robot. It is our belief that the combination of a lower field of view camera and a lidar system is a fairly common sensor configuration for most navigation systems.

For this input configuration, the following things should be considered :

- *Camera pixel dimension:* Greatly determines the number of parameters in the network. Therefore it should be kept small. However, a limited number of pixels is not able to capture more detailed features in an environment.
- *Camera field of view:* Determines how much of the environment is observable. This leads to more input data for a constant image quality but can help to obtain a better state representation since observability is better.
- *Laser sample points:* Much like the pixel dimension, the number of sample points is very important since the laser data goes trough a convolutional network to detect features. Therefore, it should provide decent resolution.
- *Laser range:* The maximum range of the laser sensor determines how far away the objects should be detected.

The hardware specifications can impact performance but do not necessarily affect the working principle of the state representation. Since the input is processed by a neural network, the algorithm is not really fixed to specific hardware. As long as each relevant state has its own unique input configuration, any hardware specification or even type of sensor should work with the state representation learning method. To guarantee this, the pixel dimension or the number of laser points should be large enough to ensure that relevant features can be extracted, but should not be too large to keep the number of training weights minimal (mitigated by downsampling).

Kinematics

Another reason for choosing the Turtlebot is that the kinematics are quite simple. The Turtlebot 3 employs a differential drive steering mechanism which makes it possible to rotate on the spot, meaning that the robot can be easily maneuvered. This makes the task easier for the reinforcement learning network since the desired pathing is not restricted by the kinematics of the robot.

Control

The agent is controlled through an action generated by the reinforcement learning algorithm based on the state prediction resulting from the current observation. This action is a velocity command consisting of a desired translational and rotational velocity sent to a low-level controller as seen in Figure 3.2. This low-level controller uses a model of Turtlebot's kinematics in order to obtain the wheel velocities such that the desired translational and rotational velocity are achieved. The velocity command from the reinforcement learning algorithm can either be a continuous or a discrete subset of predefined velocities. By sending a velocity command instead of directly controlling the speed of the wheels, the reinforcement learning algorithm does not have to learn the corresponding kinematics as a part of its solution. Since the kinematics are known and can be estimated quite accurately, it is preferred to leave this to a low-level controller. The desired wheel velocities can be directly sent to the motors, this is regulated by an internal feedback loop.

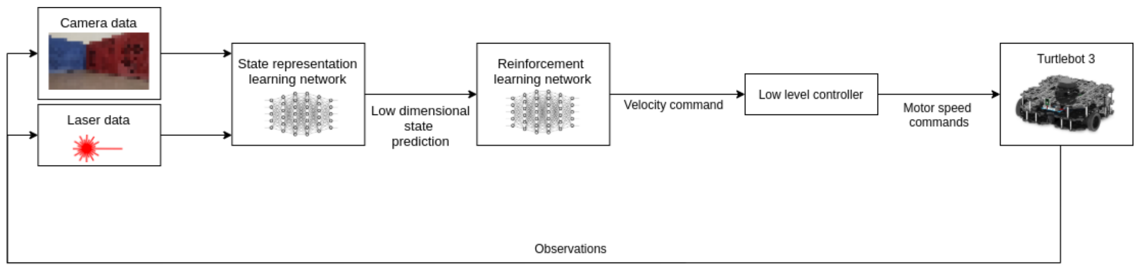

Figure 3.2: Front to end control structure.

3.3 Methodology

The goal of this thesis is to perform research in mainly two branches regarding state representation learning. One branch focuses on finding the optimal state dimension and the physical interpretation of the states. The second branch focuses on a feasibility study towards real setups. A number of experiments are performed in simulation and some are performed on a real setup. Therefore, two experimental environments have been developed.

- *Simulation environment:* A simulation environment is set up that tries to mimic the real world. It is able to capture simulated camera and laser images and is able to simulate the turtlebot's model and kinematics. Gazebo was chosen for this. This is a simulation package that works well with ROS. Gazebo is a well established open-source software package. This means that a lot of code is openly available including a model of the turtlebot 3 that is needed for the experiments. Environments are easily configurable and the robot is easy to control using ROS middleware.

- *Physical setup:* Besides that, a physical setup and testing environment is developed in order to perform the experiments regarding transfer learning. Since it was not known how the method would perform, it was chosen to start as simple as possible and proceed from there. In order to be flexible, we use colored boxes to create the desired shape.

Besides testing environments, certain software has been developed that implements the reinforcement and state representation learning algorithm and is able to communicate with the mobile robot using ROS. At the start of this thesis, a considerable part of the state representation learning algorithm was already implemented. The code was written in python, using the Theano [31] library. This led to several issues. Therefore, the algorithms could not have been tested yet. During this project, the Theano library is replaced by Tensorflow in order to solve these issues and to provide additional necessary features.

In order to answer the research questions, we need to address how performance is measured. Below, we define several performance indicators that can be used to compare results.

- *Crash ratio:* The easiest way to assess the agent's performance is by looking at the crash ratio. The crash ratio is a scalar value and is defined as the ratio between crashes and the total number of episodes. The crash ratio can be used during training to see how the learning progresses but can also be used during testing stage where it shows how robust the solution is and how capable it is dealing with noise.

- *Convergence rate:* The time it takes for the algorithm to find a solution. This can be determined by looking at several parameters. The easiest example is by looking at the crash ratio. If the crash ratio is converging, then we know that a solution is found. However, since the graph is asymptotically converging to zero, it is sometimes preferred to look at the rewards because it supplies a more direct indication of the performance. This can also show if the policy still improves while already reaching the target.

- *Quality of learned trajectory:* For navigational problems, the learned trajectory is very important. Most of the time, optimal behavior is defined as following the shortest path to the goal. The performance can then be indicated by the length of the followed trajectory. Other times it could be important to avoid an obstacle. By visualizing the followed trajectory we can see if the agent is performing the desired behavior.

- *Total reward:* Optimal behavior is defined by following the path that leads to the maximum cumulative reward. Therefore it is logical to visualize the performance by the accumulated reward per episode.

In the next chapter, it will be discussed how each of the experiments is set up in order to answer the research questions.

3.4 Summary

The first part of this chapter provided an in-depth analysis regarding the research questions. The expectation of a state representation with a too small dimension is that the performance suffers significantly, given that not all relevant data can be captured. Also, it was discussed how a too large state representation would fail to solve the curse of dimensionality and therefore would not benefit the reinforcement learning problem. Regarding the second research question, we emphasized the need for transfer learning, as it can greatly reduce operating costs in robotics. Furthermore, both the reinforcement learning method and the state representation learning method were analyzed. In this section, we discussed several parameters and their influence. For the state representation learning method, we detailed that the unsupervised learning problem can be solved by introducing a loss based on prior robotic knowledge. It was also discussed how two of these prior-based losses were adapted to use rewards rather than actions. Finally, a methodology was proposed on how to answer the research questions. This included the development of a simulation environment and a physical setup and the development of certain software.

4 Design and implementation

In this research, both the simulation experiments and real experiments are performed on the turtlebot 3. These experiments consist of navigational tasks in different setups and using different configurations. This chapter discusses the design of the experimental setups and explains how the experiments can be carried out.

4.1 Algorithm implementation

In Chapter 1, we briefly pointed out the general structure of the proposed method. In this section, we feature the algorithms, touch upon several supplementary methods, and discuss the neural network structure.

4.1.1 Reinforcement learning

The purpose of the reinforcement learning algorithm is to choose an optimal action based on the state prediction. As aforementioned, the reinforcement learning method of choice is deep double Q-learning. The basic idea of double Q-learning was pointed out in Section 2.1.4 and the concept of deep reinforcement learning in Section 2.1.5. Deep Double Q-learning is a combination of these two.

In order to address the exploration/exploitation problem described in Section 2.1.7, the ϵ-greedy method is employed. It is chosen to have a fixed ϵ. The reason for this is that since the state representation is constantly changing, it would be required to constantly reset ϵ after the state representation was updated. Besides that, preliminary tests showed that the decay of epsilon has to be tuned for each environment.

For learning the network, we make use of experience replay (Section 2.1.6). By saving experience samples and randomly replaying one or more samples each time step we break the temporal correlation between the samples and reduce forgetting issues of the neural network. The samples are picked randomly out of a list of the last n experience samples. The network is then updated using a predefined amount of samples from this list, based on the batch size.

Another consideration is how the actions are executed. In the initial code, the definition of an action was as follows: the reinforcement learning algorithm chooses one of the predefined actions in a finite set of actions. These actions consist of a desired transversal and a desired angular velocity. The actions were then performed until the desired velocity was within a certain reach from the desired velocity, therefore, we refer to this method as the setpoint-error method. This means that the duration of a time step was based on how fast the controller is able to reach the desired velocity. This was impacting the performance of the state representation learning networks since many of the priors are based on the repeatability and the predictability of an action and the respective change in state. Therefore, the method of taking an action is changed. In the new type of action, we maintain the setpoint for a certain duration instead of stopping at a setpoint-error threshold, thereby reducing the action noise immensely. In Chapter 5 it is shown how this impacts performance.

Below, the implementation of the reinforcement learning method is documented. Note that the Q-value now is based on prediction $\hat{s}_t$ rather than true state s_t.

Algorithm 3: Reinforcement learning implementation.

```
Initialize $Q(\hat{s}_t, a)$ with random weights
Copy the model network into the target network
    $Q' \leftarrow Q$
for each episode in total episodes do
    Initialize $s_0$
    Predict $\hat{s}_0$ from $o_0$ using state representation mapping $\hat{\phi}$
    for each time step t in total steps do
        $p \leftarrow$ random value [0..1]
        if $p > \epsilon$ then
            Select action $a_t$ using policy $\pi(s_t)$ according to $Q(\hat{s}_t, a)$
        else
            Pick random action $a_t \in A$
        end
        Take action $a_t$ and observe $o_{t+1}$ and reward $r_{t+1}$
        Predict $\hat{s}_{t+1}$ from $o_{t+1}$ using state representation mapping $\hat{\phi}$
        Store experience sample $[\hat{s}_t, a_t, r_{t+1}, \hat{s}_{t+1}]$
        for samples in batch size do
            Pick random experience sample $i$ from the replay buffer
            Calculate TD target
                $Q^*(\hat{s}_i, a_i) = r_{i+1} + \gamma Q'\left(\hat{s}_{i+1}, \underset{a_{i+1}}{\operatorname{argmax}} Q(\hat{s}_{i+1}, a_{i+1})\right)$
            Update network based on squared TD error using the Adam optimizer
                $E = (Q^*(\hat{s}_i, a_i) - Q(\hat{s}_i, a_i))^2$
        end
        if $\hat{s}_{t+1}$ = terminating state then
            Break
        end
        $\hat{s}_t \leftarrow \hat{s}_{t+1}$
    end
    if target network update then
        Update target network
            $Q' \leftarrow Q$
    end
end
```

Firstly, the network weights are initialized randomly, distributed and scaled according to the He normal method [32]. Then the model network is copied into the target network. Each experiment is set for a specified number of episodes. An episode starts when the agent starts from its initial position s_0 and ends when the agent reaches a terminating state or exceeds the maximum amount of steps in one episode. A terminating state means that the mobile robot is either hitting an obstacle or reaching the target position. At each time step, the reinforcement learning algorithm decides what action should be taken. Firstly the ϵ-greedy method is accounted for. If the action is to be taken randomly, an action is randomly selected from the predefined action set. If the action is deterministic, the action is decided by the Q-network. For each possible action in the current state, the network predicts a Q-value. The action that represents the highest Q-value is selected.

This selected action is then performed for a specific duration. After this time step is performed, a new observation sample is retrieved and a reward value is given based on the reached state. A new experience sample is stored in the replay buffer consisting of the initial state, next state, the taken action, and the reward value. If the buffer exceeds the maximum value, the oldest sample is thrown away. Next, the reinforcement learning network is updated. This is done using the experience samples stored in the buffer. A random sample is chosen and the network is updated using the square difference between the target Q-value and the predicted Q-value. This sequence is performed for a number of samples, based on the specified batch size.

Since we employ double Q-learning, we have both a model and a target network. The target network should slowly take over the value of the model network. In this case, we choose to copy the model network weights into the target network periodically. This is done after each episode, which corresponds to about 40-100 time steps on average.

During experiments in more complex environments, we noticed that the ϵ-greedy policy by itself could not provide desired results. Many times the agent would crash into the same spots every time, causing under-exploration in the harder to reach areas. To overcome this problem we propose an obstacle avoidance method to ensure that the agent does reach more of the environment. The algorithm will make sure that when a random action is selected according to the ϵ-greedy method, the agent cannot take any action that will lead to a possible crash. The algorithm is very simple and is documented below:

Algorithm 4: Obstacle avoidance method.

```
$X_{front} \leftarrow \min(X_{laser}[-5:5])$
$X_{left} \leftarrow \min(X_{laser}[-10:-5])$
$X_{right} \leftarrow \min(X_{laser}[5:10])$

if $X_{front} < 0.7$ then
    if $X_{left} > X_{right}$ then
        return $a_{turn\_right}$
    else
        return $a_{turn\_left}$
    end
else
    return random action $a \in A$
end
```

To implement the obstacle avoidance we make use of the laser data X_{laser}. We identify three scan values: the front scan value X_{front} is defined as the minimum value of the ten distance values in front of the agent. The left scan value X_{left} consists of the minimum of the left five distance values next to the front scan range. The right scan value X_{right} is the minimum of the five distance values right to the front scan range.

Each action step we compare the front scan value to a certain threshold, which is 0.7 meters. If the value is larger, we consider the agent free to move and to pick a random action from the action batch so it can explore. If the value is smaller, we identify a possible crash. To evade a possible collision we steer the agent either to the left or right. The agent will rotate in the direction that leads to the shortest evading move (depending on angle of approach). This ensures that the agent continues in the direction that it is heading. Since the angle of approach is affected by the ϵ-greedy method, a decent level of non-prioritized exploration is still ensured.

Next, we assign parameters that are used for the reinforcement learning algorithm during the experiments. Unless stated otherwise, the experiments are performed using the following values:

- *Learning rate (α)*: 0.001
- *Exploration rate (ϵ)*: 0.35

- *Regularization rate* (λ): 0.003
- *Batch size*: 256
- *Experience buffer size*: 10000
- *Discount rate* (γ): 0.97
- *Target network update frequency*: each episode
- *Actions size*: 3
- *Actions set*:
    1. $[v = 0.15m/s, \omega = 0.0rad/s]$
    2. $[v = 0.0m/s, \omega = 0.15rad/s]$
    3. $[v = 0.0m/s, \omega = -0.15rad/s]$

In no way, we claim that the values are optimal for the performed experiments. In this research, the scope lies more in the state representation learning method. We assess performance based on relative results.

With the exception of experiments regarding RQ1.3, the reward function is defined as the following:

$$r(s) = \begin{cases} 20, & \text{if reach destination} \\ -20, & \text{if crash} \\ 1 \text{ - e}^{(a*distance\_to\_reward)} & \text{otherwise,} \end{cases} \tag{4.1}$$

where a is a scaling factor that can be used to scale the distance from the reward. This makes sure that the proportion between the terminating states and the distance-to-target rewards can be kept similar between environments that have different sizes and therefore different maximum distances from the target. The exponential factor ensures that the robot gets encouraged to get closer to the target position.

Next, we consider the network structure of the reinforcement learning method. The reinforcement learning network should be able to map the state representation input to a set of Q-values corresponding to the action set. The neural network should act as a function approximator. The network structure (Figure 4.1) consists of an n-dimensional input layer (based on selected state dimension output), two dense layers and an output layer with three outputs that correspond to the action set.

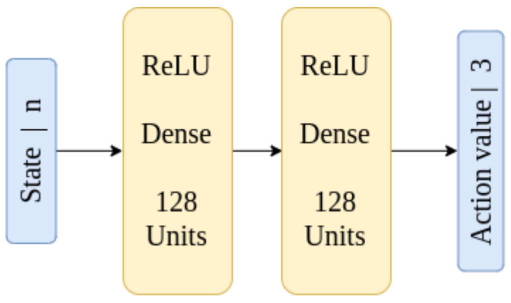

Figure 4.1: Reinforcement learning neural network structure.

The dense layers each have 128 nodes. We found that a network with 128 nodes in each layer has a better performance compared to an earlier tested network with two 32 nodes in each layer. This can come at the cost of over-fitting but was not noticed. The purpose of the second layer is to aid in developing more complex policies. The activation layers have been chosen as rectified linear units and the optimization method is chosen as Adam.

4.1.2 State representation learning

Next we consider the implementation of the state representation learning algorithm. Similar to the reinforcement learning algorithm we store a buffer that captures the latest n experience samples. Each experience sample stores the observation, the performed action, the received reward and the time step $[o_t, a_t, r_t, t]$. By using the experience buffer we can update the state representation learning method in a similar fashion to the reinforcement learning method using experience replay, thereby breaking the temporal correlation between samples.

In Chapter 3 it has been discussed how the priors are set up and how they can be represented mathematically. The implementation of the priors is straightforward. The priors can be calculated using the equations described in Section 3.2.1 and their weighted sum represents the loss function of the neural network. This loss is used to update the network parameters. For each training batch, the priors require a selection of sample pairs with similar rewards variations and a selection of sample pairs that have equal actions but dissimilar rewards.

The equal reward variations are selected as follows:

Algorithm 5: Similar reward variation.

```
for $i = 0$ to batch length do
    for $j = i + 1$ to batch length do
        if $\|\Delta r_i - \Delta r_{i+j}\| <=$ similarity constant then
            Append $[t_i, t_j]$ to similar reward variations list
        end
    end
end
```

We go through every sample of the experience buffer and try to find samples that have equal reward variation. If they are within a certain range, we store the sample combination. By initializing j from index i onwards and not from the start we assure there are no duplications of sample combinations $[t_i, t_j]$ and $[t_j, t_i]$.

In a similar fashion, the equal action but dissimilar rewards samples are selected.

Algorithm 6: Equal action dissimilar reward selection.

```
for $i = 0$ to batch length do
    for $j = i + 1$ to batch length do
        if $a_i == a_j$ && $\|r_i - r_{i+j}\| >$ dissimilarity constant then
            Append $[t_i, t_j]$ to same action, dissimilar rewards list
        end
    end
end
```

It is important to note that the similarity and the dissimilarity constant do not have to be equal. In our case, the similarity constant is set to 0.0001 and the dissimilarity constant is set to 0.3. These settings ensured that the size of the two data sets was similar for each batch.

The state representation network learning process (that runs parallel to the reinforcement learning algorithm) can be described as follows:

Algorithm 7: State representation network learning process.

```
Initialize $\hat{\phi}(o)$ with random weights
for each episode in total episodes do
    for each time step t in total steps do
        Predict state $\hat{s}_t$ from observation $o_t$ using state representation mapping $\hat{\phi}$
        Agent takes action according to state prediction $\hat{s}_t$
        Store experience sample $D = [o_t, \hat{s}_t, a_t, t]$
    end
    if SRL learning update then
        for batch in batch count do
            Create a batch of samples, randomly sampled and sized according to the
              training batch size
            Create similar reward variations list from batch using (5)
            Create similar action, dissimilar reward list from batch using (6)
            Calculate batch loss for each prior using (3.1), (3.3), (3.6) and (3.7)
            Calculate regularization batch loss using L2 norm for all networks weights
                $L_{\text{regularization}} = \lambda \sum_{j=1}^{n} \left(w_j\right)^2$
            Update state representation network using the Adam optimizer
                $L(D,\hat{\phi}) = \omega_{\text{t}} L_{\text{temporal coherence}}(D,\hat{\phi}) + \omega_{\text{p}} L_{\text{proportionality}}(D,\hat{\phi})$
                    $+ \omega_{\text{c}} L_{\text{causality}}(D,\hat{\phi}) + \omega_{\text{r}} L_{\text{repeatability}}(D,\hat{\phi}) + L_{\text{regularization}}(\hat{\phi})$
        end
    end
end
```

Similar to the reinforcement learning network, the weights are randomly initialized, distributed according to the He-normal method. Each time step, a state prediction is made according to the current observation. This prediction is used by the reinforcement learning network to select the optimal action. After the action has been performed, a new experience sample is stored in the experience buffer. This sequence is performed for each time step. In each of every n episodes, the state representation learning network is updated, according to the update interval. The state network is updated in batches. These batches are sampled randomly from the experience buffer. Next, we calculate the batch loss for each prior by taking the mean of the sample losses in the batch. The total loss is the weighted sum of all individual prior losses and the regularization loss. Using the loss, the weights of the network can be updated using the Adam optimizer.

The following set of parameters correspond to the state representation learning problem (unless stated otherwise):

- *Learning rate (α)*: 0.001
- *Regularization rate (λ)*: 0.003
- *Batch size*: 256 samples
- *Batch count*: 120
- *Experience buffer size*: 20000
- *Priors weighting (ω)*:
    1. temporal coherence : 3

2. proportionality loss : 15
3. repeatability loss: 15
4. causality loss : 15

- *Step interval*: 0.6 sec

The set of parameters has been initialized based on reference from literature and has been optimized by performing numerous experiments.

The neural network structure for the reinforcement learning method is significantly more complex than the reinforcement learning network. Mainly because of the introduction of convolutional layers to process the camera and laser data. The total structure is visualized in Figure 4.2.

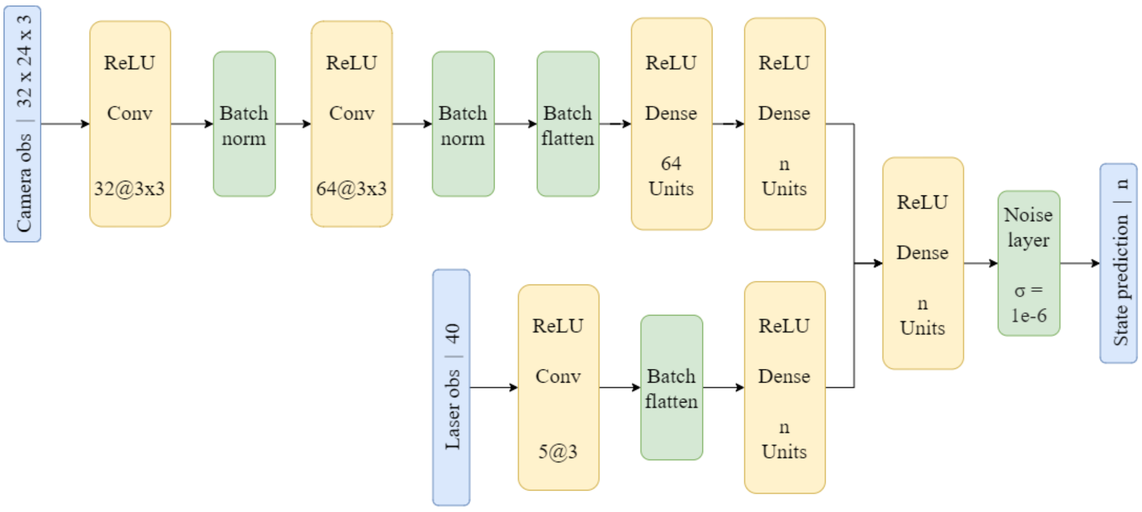

Figure 4.2: State representation learning neural network structure.

The camera data is first pre-processed by down-sampling the number of pixels to 32 by 24 by 3. The network has a parallel structure where it processes the camera data and the laser data separately. The camera data is being processed by two convolutional layers to be able to extract features from the camera images. In between the convolutional layers of the camera, batch normalization has been applied to accelerate the learning process. After the convolutional layers, the features are processed by two dense layers. The laser data is being processed by a one-dimensional convolutional layer to extract features such as corners or round objects. After that, the features are processed by one dense layer. The final layer of the camera and the laser branch both have a dimension equal to the state size. These two networks are then joined together by another dense layer. The purpose of this layer is to perform sensor fusion. Gaussian noise is added to the total outcome to prevent that two state predictions are identical. This is important since the difference between two consecutive states needs to be calculated for the loss of the priors. A difference of zero can cause problems in the calculations.

4.2 Hardware configuration

Next, we discuss the hardware setup for both simulation and real environment.

4.2.1 Simulation

Firstly we describe how the simulation platform is set up. As mentioned before, Gazebo is used as a simulation package. Gazebo is designed for simulation of robots and features a lot of readily available robotic platforms including the Turtlebot 3. The algorithms and neural networks are implemented in Python using Tensorflow. ROS is used as a middleware that serves as a communication layer between the high-level control implemented in Python and the low-level control on the robot itself. The structural overview can be found in Figure 4.3.

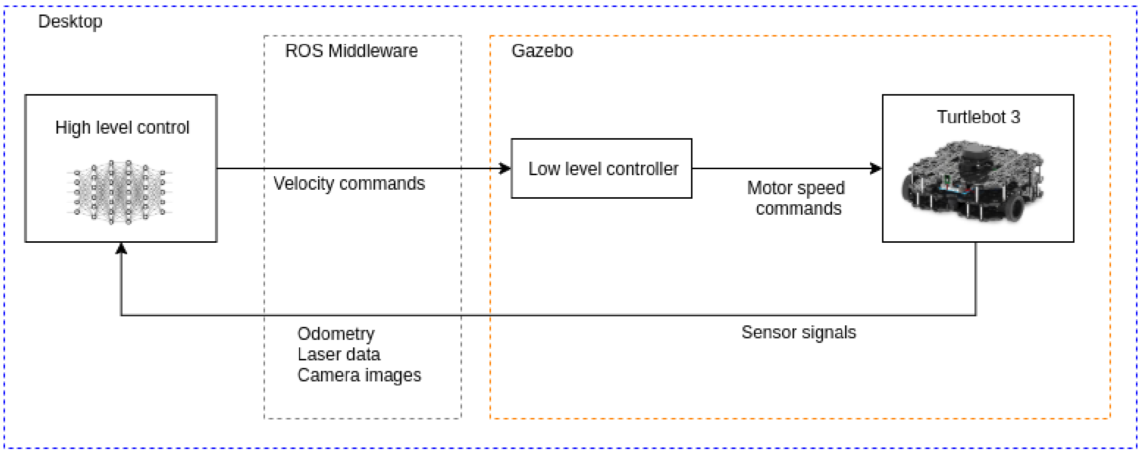

Figure 4.3: Structural overview of simulation setup.

We differentiate between two processes. A high-level control process encapsulates the two algorithms including the learning process and a low-level process that samples the sensor data and controls the motor speeds. Between these two processes lies the communication layer that is handled by ROS. The low-level process is normally located on the robot and therefore performed by Gazebo. This means that the communication structure is exactly equal to that of the real robot. In this way, the simulation is as close as possible to reality to ensure minimal transfer effort from simulation to real setup.

The Turtlebot 3 Gazebo package is distributed by one of the original developers called Robotis [33]. The accuracy of the dynamical and kinematic model is sufficient for the purposes of this thesis. The sensors are configured to match the specifications of the real robot.

- *Camera pixel dimension:* 680x480x3 pixels
- *Camera field of view:* 1.056 radians
- *Laser sample points:* 720
- *Laser range:* 5.0 meter
- *Laser field of view:* 6.28 radians

As discussed before, the camera images are down-sampled before entering the neural networks to 32x24x3 pixels and the laser data is down-sampled to 40 points.

4.2.2 Real setup

The experiments on the real robot are performed on the Turtlebot 3. The name of the specific version used is "waffle pi". The on-board computer is a Raspberry pi. The standard version includes a camera and lidar. Therefore, no additional hardware is required to run the experiments. Important hardware equipped on the Turtlebot 3 is listed below:

- *Motors:* Dynamixel (XM430-W210-T)
- *Camera:* Raspberry Pi camera module V2
- *Lidar:* Laser distance sensor (LDS-01)

- *On-board computer:* Raspberry Pi 3

As previously mentioned, the interface for the real robot is exactly equal to that of the simulation. Therefore, the structural overview is very similar to that of the simulation:

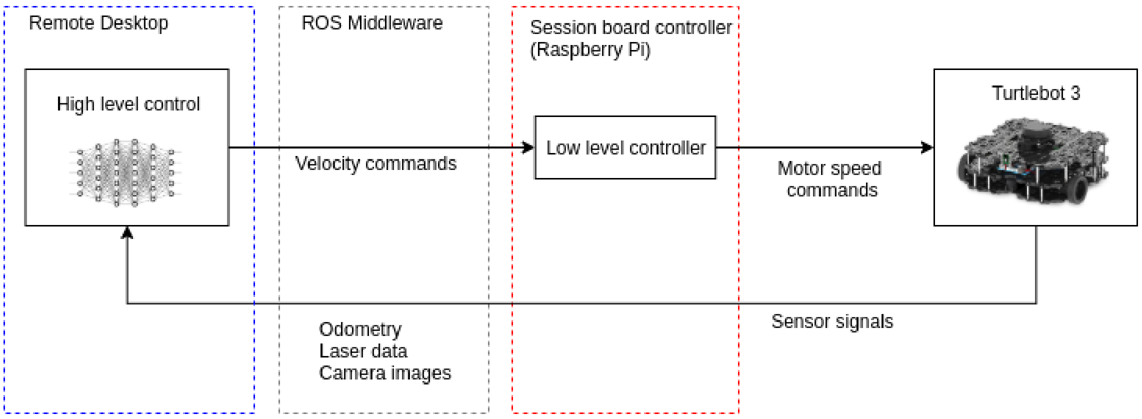

Figure 4.4: Structural overview of real setup.

The Gazebo simulation is replaced by the real robot. The low-level processes are now run on the on-board computer of the Turtlebot. This is to prevent the low-level control signals suffering from communication delay. The higher-level control is less prone to delay. This allows us to run them on a far more powerful remote computer since these processes require significant computational power. ROS can make use of the TCP/IP protocol. This means the communication between the board computer and the remote desktop can be provided through a wireless network connection. The Raspberry pi can be set up as a Wifi-hotspot to where the remote desktop can connect. The TCP/IP protocol is not designed for real-time communication. However, since the high-level control commands are not time-critical this is not a problem.

4.3 Experiment design

In this section, we will feature the different environments used in the experiments. We distinguish between mainly six environments, differentiating in size, complexity and their features. All of the environments are considered to be fully observable.

4.3.1 RQ1

The first environment is similar to that presented in Jonschkowski et al. [1]. During the project, this served as an ideal starting point, since the environment is straightforward and simple. The environment consists of a square room with four differently colored walls with a length of 2 meters and a height of 0.5 meters. The environment will be referred to as the *"small four colored walls environment"*. The environment is depicted in Figure 4.5. The objective is to reach the position indicated by the red circle in the lower right corner. The area that is classified as the target has a diameter of 0.25 meters. If the agent reaches a position within the circle, the goal is reached. The initial position of the robot is indicated by the green square. The walls each have a different color to provide the robot with information about its orientation. Since the total environment is small and simple, the duration of these experiments is relatively short, which makes it ideal for testing. Moreover, the environment will be used for comparing different sensor configurations, action methods, and will be used to investigate the effect of the state dimension on the results.

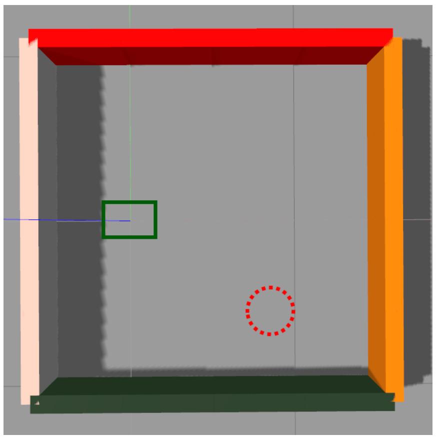
Figure 4.5: Top view of the *"small four colored walls environment"*.

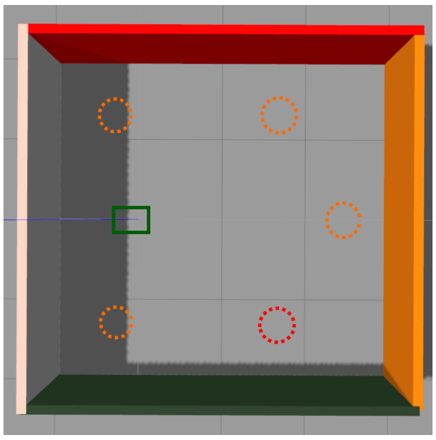
Figure 4.6: Top view of the *"large four colored walls environment"*.

The second environment (Figure 4.6) is a larger version of the first environment. The length of the walls is extended to 4 meters and the height of the walls is extended to 1.5 meters. This ensures that the walls remain in a significant portion of the camera image. The red target location corresponds to the task of the experiment designed for RQ1.2, while the orange locations together with the red location correspond to RQ2.2. This environment will be referred to as *"large four walls environment"*.

The third and fourth environment (Figure 4.7 and 4.8 respectively) have been designed to get insight into the impact of the complexity of the environment on the desired state dimension size and to see whether obstacles are encoded in the state representation.

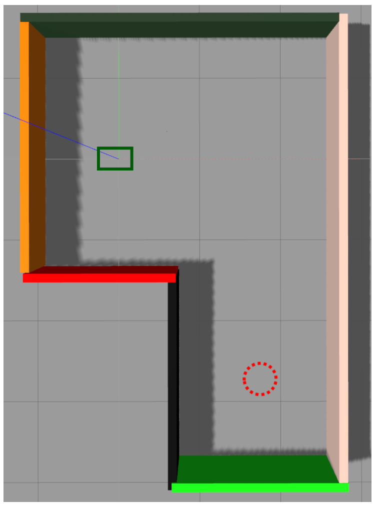
Figure 4.7: Top view of the *"L-shape environment"*.

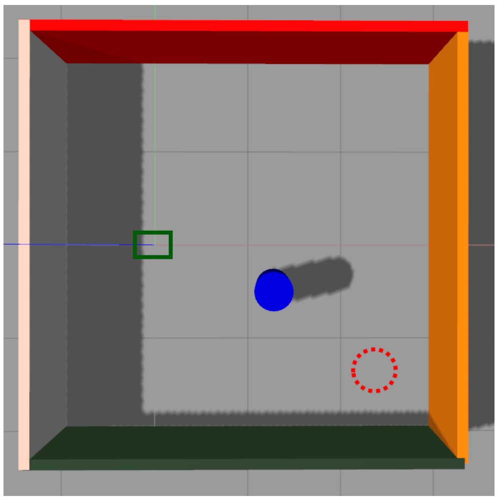
Figure 4.8: Top view of the *"Obstacle environment"*.

The so-called *"L-shape environment"* has a maximum length of approximately 3 meters and a maximum width of 5 meters. The height of the walls is 1.5 meters. The L-shape increases the complexity of the state representation and forces the reinforcement learning algorithm to make a significant turn. The fourth environment is similar to the *"large four walls"* environment, only that an obstacle (blue) has been placed on the optimal trajectory of the robot. The obstacle causes quite a disruption in the reward space and observations. Therefore, an accurate prediction of the state representation will be considerably more complex to achieve. Besides that, the reinforcement learning algorithm should be able to learn to steer the robot around the obstacle.

4.3.2 RQ2

The second research question will focus on realistic use cases. The first environment that belongs to the second research question is a simulated environment that is designed to resemble an industrial environment that has a decent amount of features. The environment has been recreated from an existing room, in which eventually tests can be performed. However, this is not within the scope of this project since the room is too complex for initial tests. By choosing this environment as our "realistic environment representation" in simulation we will already gain very useful information regarding the real-life tests. Besides that, the simulated environment can be used for transfer learning purposes.

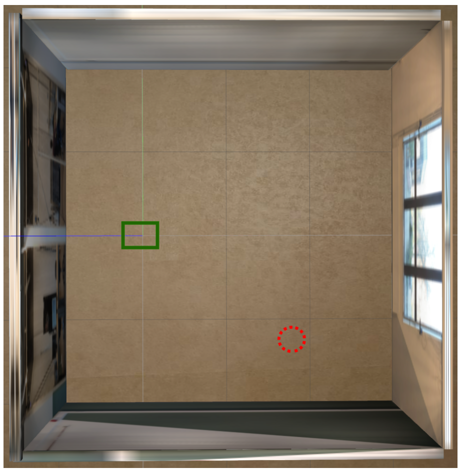

Figure 4.9: Top view of the *"realistic environment"*.

The task accompanying the first environment is visualized in Figure 4.9. The room has similar dimensions to the *large four walls environment* to allow for easy comparison. In simulation, the environment has been recreated by taking pictures of the walls and projecting them on the walls. To give an impression of how the room looks, figures of the front and right wall are shown in Figure 4.10 and 4.11 respectively. The main purpose of this environment is to see how the algorithms react on more realistic rooms and how these additional features affect learning.

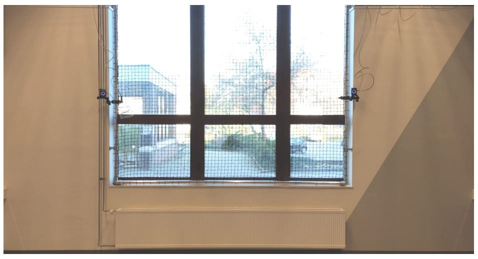

Figure 4.10: Front view of the front-facing wall in the *"realistic environment"*.

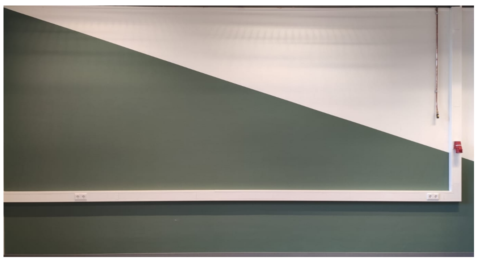

Figure 4.11: Front view of the right-facing wall in the *"realistic environment"*.

It is important to note that the room has been recreated using four flat walls. This does not account for minor deviations in distances that the lidar might observe due to objects hanging in front of the walls.

The second environment will be used to investigate the possibilities of transfer learning. As previously mentioned, the environment proposed above will not be used for this. The reason for this is that the environment is too large and has some more complex parts that need to be taken into account for transfer learning. Since it is not know how transfer learning will perform, the choice was made to start off with a very simplistic environment, similar that to the *small four walls environment*. The environment is set up using colored boxes as shown in Figure 4.12.

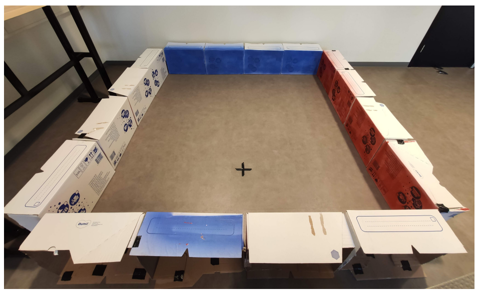

Figure 4.12: Telescopic view of the *"four walls boxes environment"* for real setup.

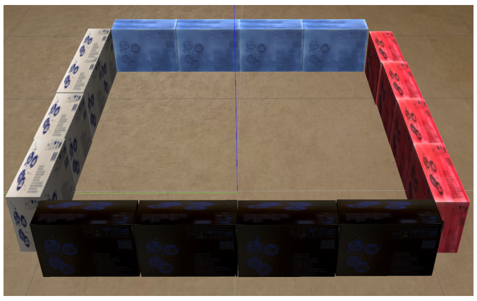

Figure 4.13: Telescopic view of the *"four walls boxes environment"* in simulation.

The environment was easily recreated in simulation by taking pictures of the front side of the boxes and placing them on the equally sized boxes in simulation. The simulated environment is displayed in Figure 4.13. The task accompanying the real environment is similar to that of the *small four walls environment*, as depicted in Figure 4.14.

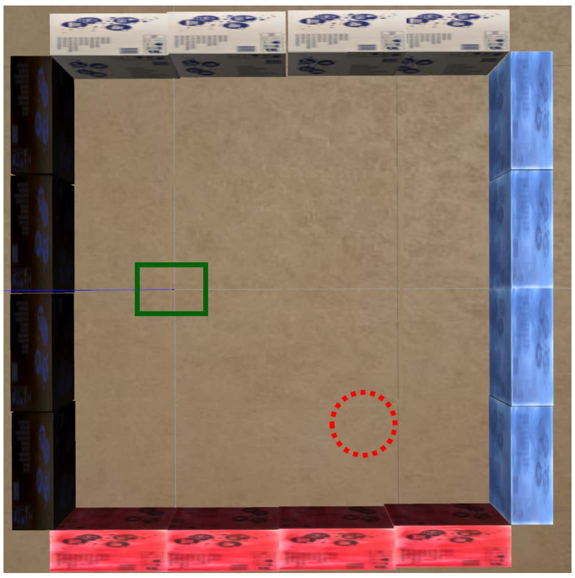

Figure 4.14: Top view of the *"four walls boxes environment"* in simulation.

The agent starts at the green square and the target location is indicated by the red circle. The task is relatively simple, however, since the robot has to take a rather sharp turn to reach the goal we can hopefully distinguish between deterministic behavior and a lucky run.

In order for the transfer learning to work, it is essential that the simulated environment represents the real environment as well as possible. To verify, we have captured the robots camera inputs for both the simulation and the real setup. In Figure 4.15, the camera input for the real setup is visualized. In Figure 4.16, the simulated observation is shown.

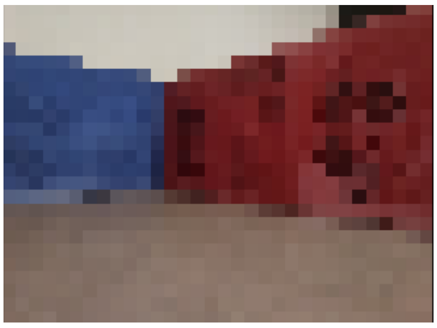
Figure 4.15: Resized camera observation for real setup.

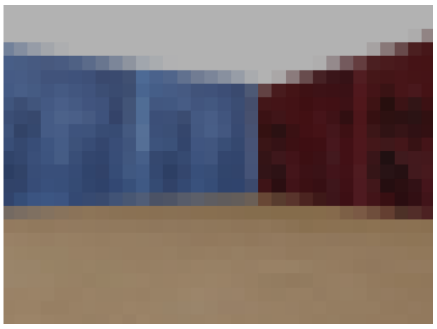
Figure 4.16: Resized camera observation in simulation.

As can be seen from the figure, the captured images are very similar. This can be attributed mainly to the reduced image size because higher-level details disappear. This makes it significantly easier to match simulation to reality. The main discrepancy between the figures is illuminance. The lighting is a very difficult part to match in simulation. Later in the document, we show how this affects results. Also, a suggestion is given on how to mitigate this problem.

4.4 Summary

This chapter described how both the reinforcement learning algorithm and the state representation learning algorithm were implemented. It discussed important aspects such as exploration, obstacle avoidance, the reinforcement learning network structure, and most importantly the state representation learning network structure. The state representation network features two independent layers that process the camera and laser data. These layers consist of convolutional layers and dense layers that are finally merged into a sensor fusion layer. This layer predicts the final state from the two independent sensor modalities. Furthermore, the hardware configuration for the simulation setup and the real setup were discussed. Our implementation allowed us to move from simulation to real robot with only minimal changes to the code. This was mostly due to the convenient message structure in ROS that is independent of the sender and receiver. Finally, we proposed a total of six environments that will be used to perform the experiments. Four of these are artificial environments that correspond to the first research question and differ in both size and complexity. Two of the environments are based on realistic use-cases and will be used for research question number two.

5 Results and discussion

The first part of this chapter describes preliminary results obtained during the debugging process. After that, a comparison between the duration-based and setpoint-error-based action is made. Next, we show the added value of using a lidar module compared to only using a camera. The remainder of the experiments is dedicated to the research questions. We feature an in-depth analysis of the state representation network in multiple environments in order to determine the optimal state dimension for the proposed method. The next set of experiments is dedicated to realistic use-cases, corresponding to research question two. Finally, we show the added value of the state representation method compared to reinforcement learning by itself. The main performance criteria that will be used are the crash ratio and average episodic rewards during training since they indicate the learning speed and the quality of the policy. In these experiments, we will make use of the environments proposed in Chapter 4.

5.1 Initial results

In this section, we will discuss the initial set of results. These preliminary results document the debugging and optimization process that has been performed in the first stages of the project.

5.1.1 Testing the state representation learning network

During implementation phase, the first step was to provide a working state representation learning algorithm. Even though the state representation learning network is dependent on reinforcement learning generated results, the state representation learning network can be tested independently by generating a data set beforehand with an agent that performs random actions. The experiments were conducted in the *small four walls environment*. The dimension of the state representation was five. In total, the agent performed 400 episodes. The randomly taken actions caused the agent to perform a movement pattern similar to that of a Gaussian distribution as shown in Figure 5.1. We can see that the trajectories are nicely distributed, which is really helpful to form a good initial state representation. The observations sampled on these trajectories will be used to learn a state representation.

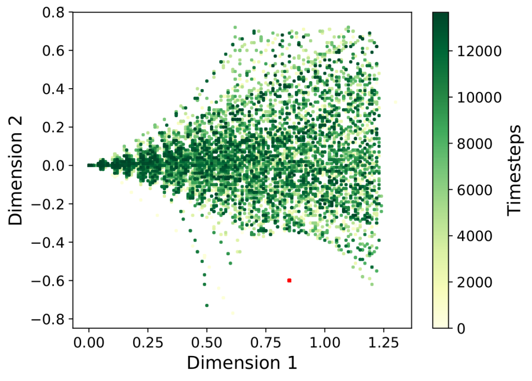

Figure 5.1: The position samples give an indication on the distribution of trajectories that result from performing random actions.

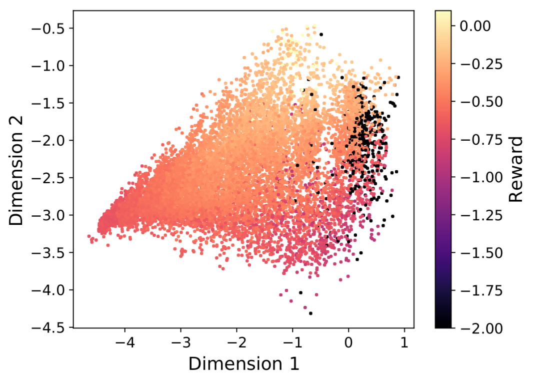

Figure 5.2: First two principal components of predicted state representation. Note that this is a two-dimensional representation of five-dimensional data.

In Figure 5.2, the initial results obtained by training the state representation network on the samples generated by random actions are shown. The results describe a two-dimensional principal component analysis of the states, meaning that we plot the specific planar projection of the five-dimensional state representation for which the most variance is explained. The samples are colored based on the reward obtained by reaching a specific position. The +20 and -20 rewards received from reaching terminating states are clipped onto 0 and -2 respectively to be able to distinguish colors between the other samples.

The advantage of PCA is that we can analyze the five-dimensional data using a two-dimensional plot. Principal component analysis can help us observe the clustering and disentanglement properties of the network. However, it should be noted that the mapping is linear and therefore can fail to show non-linear relations/clustering in the state representation.

From the right figure, we see that equal colors are placed together and that the transition between them is smooth. This means that the network is able to place similar rewards together and that there are no discontinuities in the reward space (as it should be) except for reaching the target or hitting an obstacle. The placement of the rewards can also tell something about the quality of the state prediction. On top of the figure, we see rewards of 0 that are clustered. This corresponds to the location of the target. Samples further from the target gradually decay in terms of reward. This means the structure of the environment in terms of distance is captured well since the rewards are based on the distance from the target. Finally, we can comment on the black dots. These samples correspond to the agent hitting an obstacle. These points are all placed at the outsides of the state prediction, similar to the environment it tries to encode.

5.1.2 Testing the reinforcement learning network

In order to test the reinforcement learning implementation, the reinforcement learning network was supplied with ground truth input. This ground truth data vector included the x-position, y-position and the cosine and sine of the orientation. In this way, we can fully represent the angle without the shift that normally occurs in a radial representation. This otherwise discontinuity is expected to cause problems in the network. The double Q-learning algorithm was tested on the *small four walls environment* for 500 episodes. The target was situated in the right lower corner. As can be seen in Figure 5.3, the reinforcement learning method is able to obtain a policy that is able to reach the target very quickly. Considering the very small amount of trajectories that are not going in the direction of the target, we can see that the reinforcement learning algorithm learns very fast and is well-tuned. Since epsilon is constant and set at 0.35, we can see that the mean of all trajectories is more or less the ideal trajectory, accompanied by a significant amount of variance that is caused by the ϵ-greedy method.

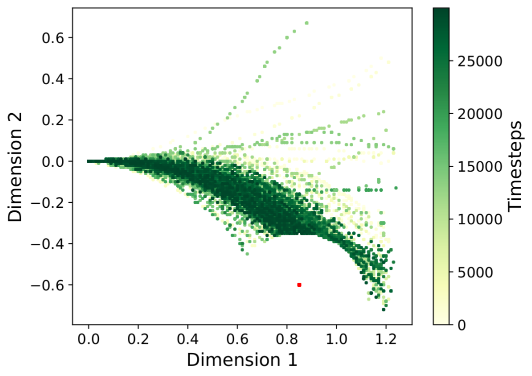

Figure 5.3: Position samples show that the reinforcement learning method is able to find a good policy based on ground truth input.

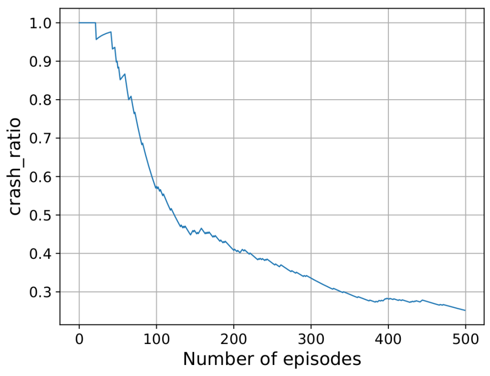

Figure 5.4: Crash history of reinforcement learning algorithm based on ground truth shows rapid convergence of policy.

As can be seen from Figure 5.4, after 30 episodes the agent is already able to find the target and therefore reduce the crash ratio. At the 120 episode mark, the crash ratio seems to stagnate slightly and seems to deviate from a perfect asymptotic shape that would describe reaching the target every time. This might be explained by the fact that the network tries to find an ideal trajectory where it has to rotate the least amount of times and still be able to reach the target, since every excess rotation count as an extra time step and therefore decreases the total return. This can also be seen from the position graph since it shows that after a while the agent slightly misses the reward and eventually crashes into the wall. At time-step 400 we see another slight increase in the crash ratio that can be explained in the same manner. Overall, it can be concluded that the reinforcement learning algorithm is able to perform very well in this environment.

To further evaluate the reinforcement learning algorithm, the Q-value value assigned by the reinforcement learning method has been visualized for each of the positions that were visited by the agent. In Figure 5.5 it can be seen how the network attributes positions close to the target a high Q-value and the positions further away with a lower Q-value. Samples that do not lead to the target are given a very low maximum Q-value. This further validated that everything was working accordingly.

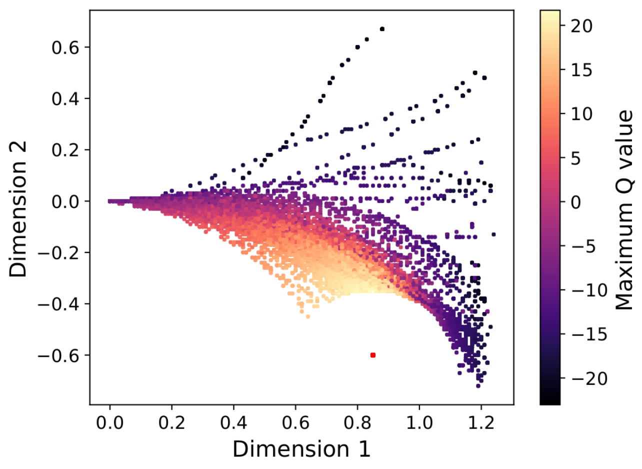

Figure 5.5: Maximum of Q-value prediction for each position sample. This shows how the agent "thinks" and allows us to see the ideal trajectory according to the agent.

5.1.3 Combining the two methods

To verify whether the state representation network can work along with the reinforcement learning network, we now use the state prediction instead of ground truth input. The state representation learning method was updated alongside the reinforcement learning method, as described in Chapter 4. The reinforcement learning method was updated every time step using experience replay and the state representation learning method was updated every 250 episodes using a five-dimension state representation. From Figure 5.6 we see that initially the average episodic during the first 250 episodes was around -30. This corresponds to the agent exploring the environment randomly. At the 250th episode, we see a large downwards spike. This corresponds to the first time that the agent was following a (bad) policy. During the next 250 episodes, the agent was still not able to find a decent policy since the state representation is still under-developed. After the 500th episode we see that the agent is starting to achieve a higher average reward since state representation is now a lot better. At episode 1000, the agent was able to find the target consistently. Slightly after every state representation network update, we notice a significant downwards spike. This is caused by the fact that the reinforcement learning network needs to adapt to a new state representation. After the 1500th episode the agent starts to perform worse. This was caused by the fact the state representation becomes more complex and harder for the reinforcement learning network to adapt to. Overall, we can see that the state representation learning method performs significantly worse compared to using ground truth since ground truth adapted a policy in under 50 episodes as shown in Figure 5.4. This is expected since we now make use of camera and laser data to obtain a policy. Ground truth data is far superior to this.

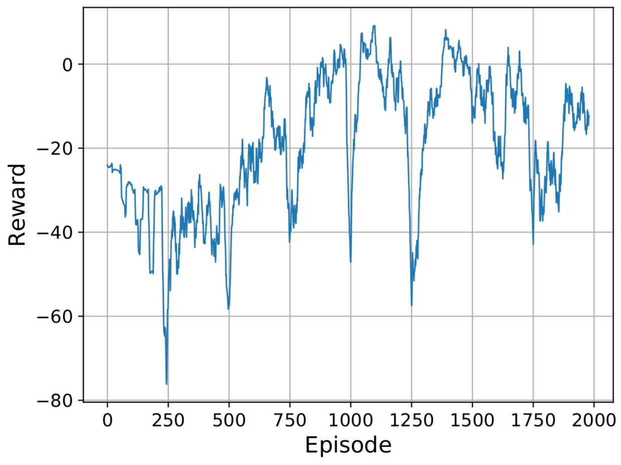

Figure 5.6: Moving average (20) episodic reward of state representation learning and reinforcement learning combined. The state representation updates are nicely captured by the "dents" in the rewards since this causes the reinforcement learning network to adapt to an updated state representation.

In order to understand what is going on inside the "brain" of the agent, we can see how it perceives the environment. The state representation predicted by the network is shown in Figure 5.7. By analyzing the state representation it can be seen that the initial position of the agent corresponds to location $[-2, 1]$ from the clustering of low rewards and the point-like shape. The target is located at the top of the graph as this is where the high rewards are clustered. These locations will be important for the upcoming analysis.

To get an idea of how the reinforcement learning method maps the state prediction to Q-values we will consider Figure 5.8. Using this graph, we can see how the reinforcement learning network estimates the Q-value based on the state prediction that is made from the camera and laser images. Just before, we located the initial position and the target. What we see is that the Q-value starts low at the initial position and gradually increases as we move to the target. The other samples are outside the ideal trajectory and are given lower Q-values. This graph gives us an indication of how the agent "thinks" and allows us to give us a view inside the "black box" of neural networks. The plots give us an indication that the method works as expected.

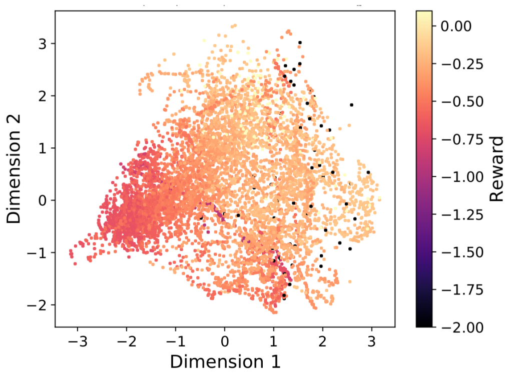

Figure 5.7: First two principal components of predicted state representation colored by corresponding reward obtained in experiment.

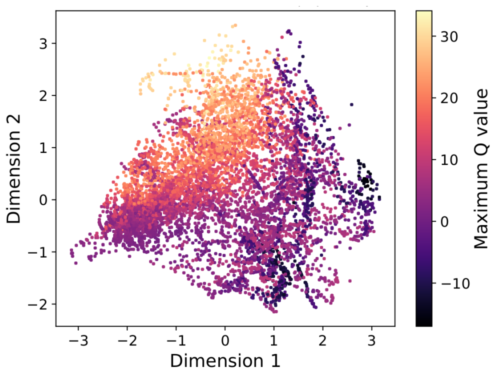

Figure 5.8: First two principal components of predicted state representation colored by predicted Q-value by reinforcement learning network.

In the previous figures, we described the mapping from state prediction to Q-value. We can go one step further by showing the mapping between the observations obtained at certain position samples and their maximum Q-value (Figure 5.9). This plot is similar to that of the ground truth reinforcement learning network (Figure 5.5). The difference is that that the prediction now has to be made using both of the networks: *observation* $\xrightarrow{srl_{network}}$ *state prediction* $\xrightarrow{rl_{network}}$ *Q-value*.

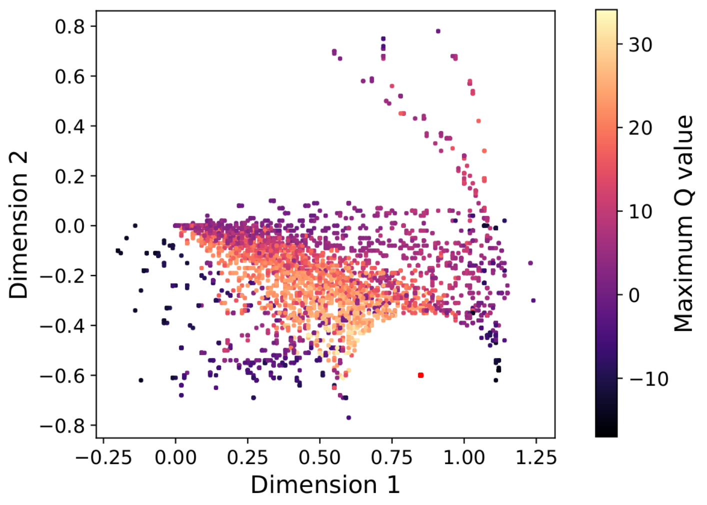

Figure 5.9: Position samples colored by predicted Q-value give us insight on how the two networks perceive the environment.

The initial position of the agent can be easily located since we know this is at [0,0]. The target location is described by the red dot. Similar to Figure 5.8 we see a gradually increasing Q-value between initial position and reward. Trajectories that do not lead to reaching the target are given very low Q-values as expected. These graphs are a very powerful debugging tool since they can visually represent the obtained policy and show the mapping of the two networks together.

5.1.4 Setpoint-error-based actions versus duration-based actions

In Chapter 4 it was mentioned how the action methodology was changed to perform duration-based actions rather than setpoint-error-based actions. To compare the two methods, two experiments have been set up that both make use of a different action method. The experiments are located in the *small four walls environment*. The state representation network is updated during the 250th, 500th and 750th episode. After that, the reinforcement learning network is given time to find the optimal policy.

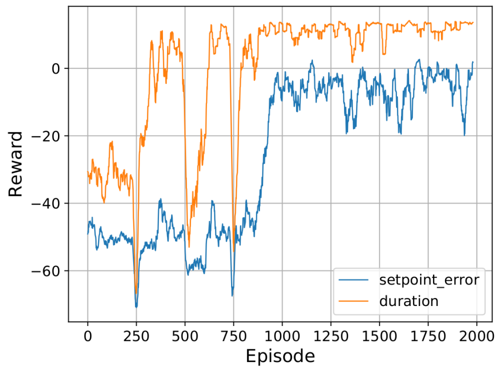

Figure 5.10: Moving average reward (20) comparison of setpoint-error and duration-based action.

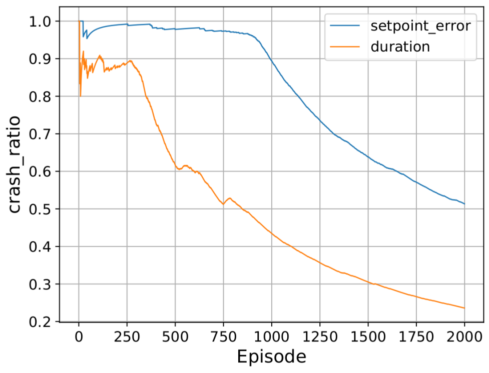

Figure 5.11: Crash ratio comparison of setpoint-error and duration-based action.

From Figure 5.10 it can immediately be seen that the duration-based method yields a significantly higher reward at the initial and the final episodes. The reason for this is that the setpoint-error method performs overall more actions and therefore suffers more from the distance to target penalty. This does not say anything about the performance. During the first 250 episodes, both methods are performing random actions. From the crash ratio in Figure 5.11, we can see that the duration-based reward reaches the target a lot more during this phase. This can cause a minor exploration advantage that has to be taken into account. After the 250th episode we can see that the duration-based method almost immediately starts reaching the target while the setpoint-error-based method performs close to random. After the second state representation learning update in episode 500, we see the duration-based method finding its target even faster. It is only after the third state representation learning update that the setpoint-error starts to find the target. The duration-based method is three times faster in terms of training speed since it is able to find the target at episode 300 compared to episode 900 for the setpoint-error method. The improved method also makes the agent reach the target more consistently. This can be seen from the reduced fluctuations in reward after the 1000th episode and can be confirmed from the crash ratio graph since it converges to a significantly higher average.

The difference in performance can be attributed to the more consistently taken actions. This benefits the reinforcement learning method, but mainly the state representation learning method. The robotic priors are largely based on the difference between the current and the next sample. If there is a lot of action noise, it becomes significantly more difficult to construct a well-defined state representation. Having more consistent actions means that a far better state representation can be structured. This is also the reason that the duration-based method can find the target after one update while the setpoint-error-based method needs to be updated three times.

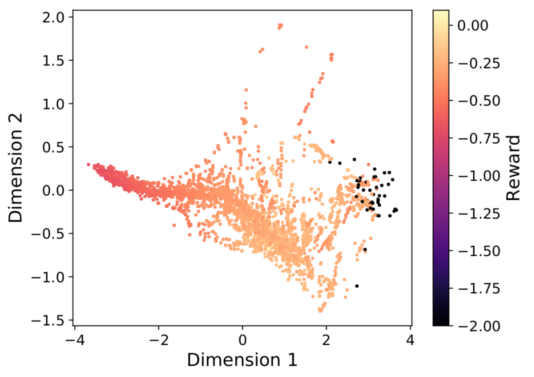

Figure 5.12: First two principal components of state representation formed by setpoint-error-based action.

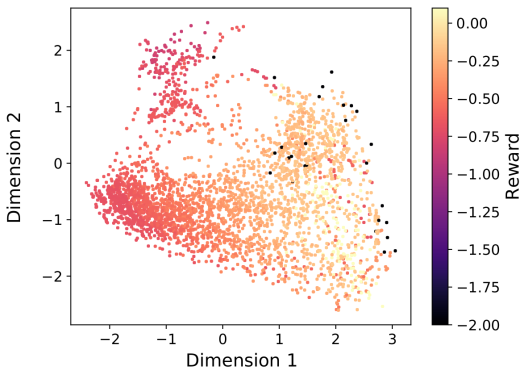

Figure 5.13: First two principal components of state representation formed by duration-based action.

By comparing Figure 5.12 and 5.13 we see that the duration-based action state representation is far less compressed. The network is able to differentiate between different states far better, which is beneficial for the reinforcement learning algorithm. For example, in the setpoint-error experiment, all terminating states are clustered (black dots). In the right graph, we see that the agent can differentiate between the locations where the agent does hit the wall. The same conclusion can be drawn from the points corresponding to the general samples since the overall shape is much broader. In the right graph, we can also identify the target location much better by the clustering of the light spots. In the left graph, this is almost unnoticeable. By increasing the consistency and repeatability of the actions we can significantly improve the quality of the state prediction and thereby significantly decrease training duration and more consistently reach the target position. From now on the experiments have been conducted using the duration-based actions unless stated otherwise.

5.1.5 Input data analysis

In Chapter 3 it was explained how the lidar sensor was selected to operate along the camera to extend the field of view. In this section, we investigate how each of the sensors contributes to the performance. The experiments have been conducted in the *small 4 walls environment*. The state representation network is updated three times with an interval of 125 episodes since the new action method allows the state representation network to develop significantly faster. In Figure 5.14, we see how the crash ratio compares for three types of experiments. One was performed with both the laser and the camera as inputs, one was performed using only the camera input data and one was using only the laser input data.

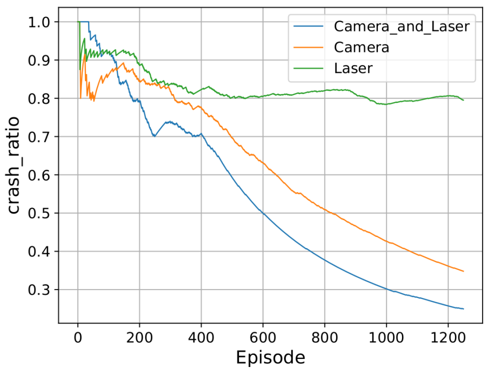

Figure 5.14: Crash ratio comparison for different sensor configurations shows that adding the lidar can improve performance of the state representation learning method compared to camera only input data.

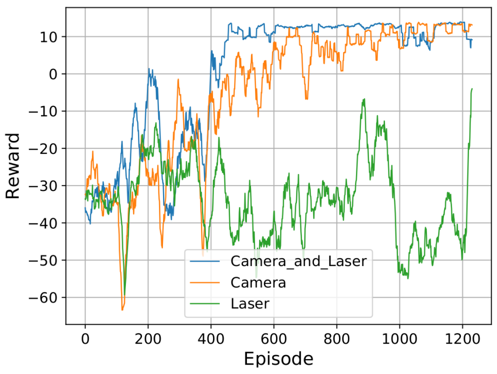

Figure 5.15: Average episodic reward comparison for different sensor configurations shows that by using only camera data the time it takes to find a final policy is much slower compared to the combined method and that the lidar sensor by itself is not sufficient to solve the task.

During the first 125 episodes, the agents perform random actions. After that, the combined sensor method starts finding the target almost immediately, while the other two methods fail to do this (Figure 5.15). After the second update, the average reward for the camera-only experiment starts to improve, while the combined approach performs slightly worse. After the 375th episode, the state networks are not updated anymore and the reinforcement learning networks are given time to develop a policy. From this point onwards, we can see that the combined method develops a good policy really fast. The camera-only method does also improve its policy, yet, this is happening very slowly in comparison. At the 450 episode mark, the training can be considered done for the combined method. The camera only method is effectively done with training at episode 1000. This means that the combined input data can reduce training time by around 50% compared to only camera input data. The method that uses only laser data is not able to find a policy. At the end of the experiment, the crash ratio shows close to random behavior. Next, we investigate their state representations.

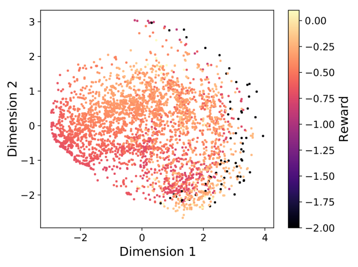

Figure 5.16: The first two principal components of the state representation formed by laser data shows that the method can not correctly cluster similar rewards.

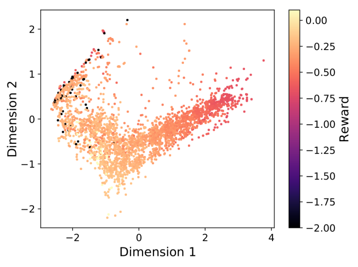

Figure 5.17: The first two principal components of the state representation formed by camera shows that the camera is much better in distinguishing rewards, but lacks distinctions between observations.

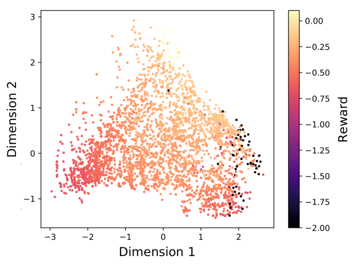

Figure 5.18: The first two principal components of the state representation formed by both sensors show correct clustering of similar rewards and also good distinction between similar but different observations.

Since the room is symmetrical in both directions, it is almost impossible for only a laser to correctly estimate the agent's orientation. Besides that, many states become ambiguous in terms of observations. This means the network has trouble differentiating between states, as can be seen from Figure 5.16. The samples that correspond to the target are not correctly clustered. Also, a lot of reward values are mixed-up. This is a good indication of a poorly described state representation. The state representation formed by the camera is already much better in terms of clustering equal rewards as can be seen in Figure 5.17. However, without the laser, the shape of the state representation is much more compressed. The state network has much more trouble making a distinction between close but not equal positions. This makes sense since it is hard to locate a position using camera data only. Distance related data is far better doing this job. The combined sensor configuration is displaying good properties of both. As can be seen from Figure 5.18, the rewards are correctly clustered in terms of colorization and the state network is able to differentiate between similar but not equal observations.

In this section, it was shown the camera and lidar can be used complementary. It was shown that the state representation using both sensors showed much better properties compared to the sensor on their own. Also, it was shown that the training duration was reduced by 50%.

5.2 Results corresponding with RQ1

In this section, results corresponding to the first research question are presented. The experiments are aimed to investigate the effect of the state dimension on training performance. Moreover, it will be investigated how different types of environments and can affect the optimal state dimension.

5.2.1 Effect of state dimension

To analyze the effect of the dimension of state predictions on the learning speed and performance, a number of experiments have been performed on the *small 4 walls environment*. For each of these experiments, a different state size was assigned. The state representation network was updated on a 125 episode interval for a total of three times. After the state representation updates, the reinforcement learning method is given enough time to learn a good policy. The duration of each experiment is 1250 episodes.

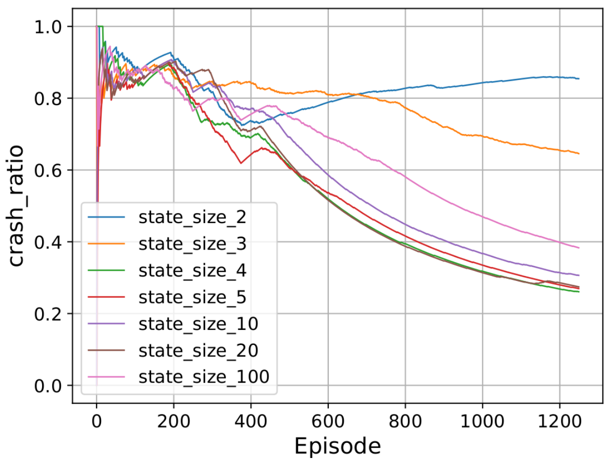

Figure 5.19: Crash ratios of different state representation dimensions show the relevance of choosing the right state dimension.

Looking at Figure 5.19, it can be seen that during the first 425 episodes all methods perform very similar. At this point, the state network is still being updated. After the 425th episode, the state representation is kept fixed to give the reinforcement learning network time to find a good policy. At this point, we start seeing differences. At the end of the experiment, we can conclude the following things: we notice that the two experiments with a state size smaller than four perform significantly worse. For both experiments, the reinforcement learning algorithm is not even able to develop a policy to find the target consistently. The experiment with state size 5, 10 and 20 perform best and are all very similar. The experiment with state size 100 performs somewhere in between the two groups.

The poor performance of the experiments with a state size smaller than four can be explained by the fact that the state representation was not able to encode all necessary information in the predictions. This lack of information causes the reinforcement learning algorithm to be unable to find the target consistently. The restriction in dimension makes it very challenging for the network to correctly map all points. How this happens, and why only for experiments with a state dimension under four, will be further discussed in this chapter.

The reason for using state representation learning is to compress the input data to a size that can be efficiently handled by the reinforcement learning network. We have just described what occurs when the state size is too small and not able to capture all information. We can also show what happens when the state dimension is too large. This can be seen from the graph corresponding to state size 100 in Figure 5.19. In this case, the reinforcement learning network receives too much irrelevant information and has trouble finding the proper correlations between input and expected return. This is what we refer to as the curse of dimensionality. While the method is still able to find a policy, it has become significantly harder for the reinforcement learning network to develop a good policy due to the abundance of irrelevant information.

These experiments have shown that the optimal state dimension is a trade-off between a decent compression of input data to speed up learning and having a state dimension that is large enough to capture all relevant data. A state dimension that is too small cannot fully represent the state, causing the reinforcement learning algorithm to be unable to find a competent policy while a state dimension that is too large causes the reinforcement learning method to suffer from the curse of dimensionality.

5.2.2 State representation analysis

In this section, we perform a more in-depth analysis regarding the state representation. This is done using the results of an experiment with a ten-dimensional state that is updated six times on a 250 episode interval. Choosing ten components ensures that the state network is not restricted. By choosing a total of six updates we ensure that we asses the state representation in a fully developed state. The experiment is performed in the *small four walls environment.* To analyze how the state is encoded, a principal component analysis is performed. By investigating the amount of variance explained by the components we can identify the actual number of components needed to describe the total state representation. Components that do not describe any variance are linearly correlated to the other states and therefore do not contribute to the amount of information in the state representation.

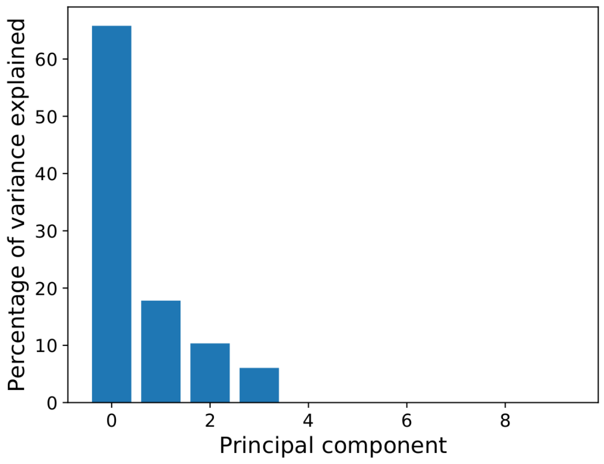

Figure 5.20: Principal component analysis of state prediction shows that four principal components describe all of the variance in the state prediction data set.

As indicated in Figure 5.20, the amount of uncorrelated components needed to describe the total variance is four. This means that the state representation can be described in terms of four parameters without information loss. These results can be linked to observations in Figure 5.19, where it was shown that state predictions lower than dimension four performed significantly worse. This can be explained by the principal component analysis described in Figure 5.20 since it shows that the minimal amount of components needed to fully describe a relevant state representation is four.

The robotic priors are largely inspired by Newtonian mechanics. Therefore, we might expect the state prediction to encode information that is related to certain physical properties (that are relevant in a two-dimensional navigation problem), such as position, orientation or distance. In order to investigate further, a correlation analysis between the first four principal components and several physical properties was performed. This was done by conducting a new experiment with a ten-dimensional state representation. The agent was set to perform random actions similar to Section 5.1.1 for 400 episodes. By choosing random actions instead of taking samples from a policy guided trajectory, we can ensure that the environment has been trained on equally. This also solves another problem, which is that taking only samples from a certain trajectory causes the physical properties to be correlated to each other. Choosing a broader set of samples can help identify single components more easily.

In this analysis, we first normalized all data to have a mean of 0 and a variance of 1 to directly obtain the Pearson correlation coefficient. The correlation coefficient can be used to compare correlations of different data sets since they are normalized. The correlation coefficient is scaled between -1 and 1. A value of 0 means no correlation, a value of 1 means perfect positive correlation and a value of -1 means perfect negative correlation. First of all, it should be noted that this new experiment using random actions resulted in only three principal components compared to four shown in Figure 5.20.

Table 5.1: Correlation coefficients of principal components and physical properties using random actions.

	X-position	Y-position	Distance to target	Orientation
Principal component 1	0.95	0.0	-0.53	-0.13
Principal component 2	-0.15	-0.88	-0.78	-0.70
Principal component 3	0.0	0.0	0.06	-0.42

From Table 5.1 it shows that there is a very strong correlation between the first component and the x-position of the agent. The second component is mostly correlated to the y-position. The distance and the orientation also show up in this component. The reason for this is that these physical properties are correlated to themselves. The action pattern in this experiment was equal to that shown in Section 5.1.1. Since the agent initially is heading towards the x-direction, the development of the y-position is highly determined by the rotation the agent makes. Therefore, these two properties are heavily linked, resulting in a correlation coefficient of 78%. The distance to the target was also found to be heavily correlated to the y-position with a coefficient of 78%. This can be understood in the same manner. The third component is mostly correlated to the orientation.

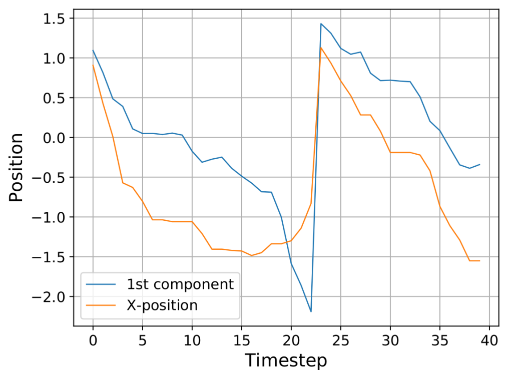

Figure 5.21: Visualization of the first principal component and the x-position (both normalized) shows a significant similarity in terms of shape.

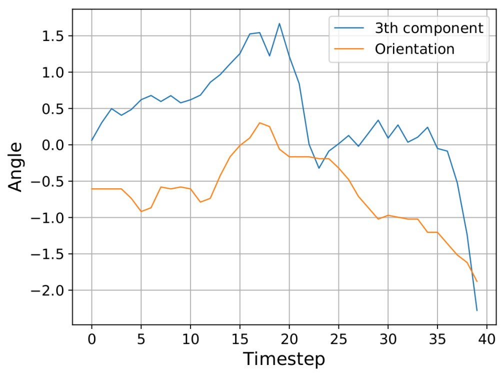

Figure 5.22: Similarity between the third principal component and the orientation (both normalized) is less strong but still present.

The high correlation between the principal components and the physical properties can also be visualized. From Figure 5.21 it can be seen that there is a large similarity between the first component and the x-position. At time step 23, the agent is reset to the initial position. This can be seen from the sudden jump in x-position. When this happens, an equal jump occurs in the samples corresponding to the first principal component. The overall shapes of the two lines are very similar and the samples corresponding to the principal component are even able to describe the small spikes that occur in the x-position. In Figure 5.22, the correlation between the orientation and the third component is visualized. This similarity is much less obvious but still present. The orientation is most likely much harder to estimate compared to the position which may explain why the similarity is smaller.

The similarities between the state prediction and the physical properties are very interesting since the network did not receive a labeled set to learn from, meaning that it can be considered as an unsupervised learning problem. The developments of these components in terms of position and orientation can be understood intuitively. In Jonschkowski et al. [1] it was shown that a similar task, where the agent could only move transversal but not rotate, only two principal components were relevant. This can be explained by the fact that only two factors are needed to describe the reward structure, namely the x- and y-position. When the agent can also start rotating, there is at least one extra component needed to differentiate between samples. The agent can take samples on the same position with entirely different angles leading to entirely different rewards or rewards variations. This means that an extra degree of freedom is needed to be able to describe these differences. This explains the development of three principal components. These components describe variables that are needed to map the reward space, in this case, the x-position, y-position and the orientation of the agent. This also shows that the network automatically finds out the complexity of the task and structures the state representation according to this.

However, this does not yet explain the development of a fourth component. The fourth component shows up after the agent starts taking deterministic actions. At this point, the trajectories become more versatile and the agent receives different observations. This set of different trajectories results in more complex observations and a broader reward space. To simulate these more versatile trajectories and to still be able to ensure all parts of the environment are mapped equally, a new experiment was performed using random action with double rotational velocity.

Table 5.2: Correlation coefficients of principal components and physical properties using random actions with double rotational velocity.

	X-position	Y-position	Distance to target	Orientation
Principal component 1	0.80	0.02	-0.27	-0.12
Principal component 2	-0.03	-0.81	-0.81	-0.74
Principal component 3	-0.39	-0.25	0.17	-0.43
Principal component 4	-0.08	-0.11	0.01	0.24

The doubling of the rotational velocity gave rise to a fourth component (although relatively small compared to the others). The first and second components had to be rotated by 45 degrees because the position samples are now more distributed throughout the environment, meaning that the most variance now occurs diagonally as shown in Appendix A. Looking at Table 5.2, the structure is still close to that of the results in Table 5.1. The first component corresponds mostly to the x-position, the second component is highly correlated to the y-position, distance, and orientation, and the third component is again mostly correlated to the orientation, although, it also correlates to the X-position very strongly. The fourth component correlates most significantly to the orientation.

In this experiment, the extra component is mostly describing the orientation. During multiple experiments, it was found that the structure of this fourth component was not always consistent. Therefore, it cannot be said that the extra component describes one property specifically. The most likely explanation for this extra component are the samples corresponding to the left and right walls. As can be seen from Appendix A, the mobile robot hits the left and right wall much more whenever the angular velocity is increased (or when the robot starts following a sub-optimal policy). The samples corresponding to hitting an obstacle are quite harder to map since the reward value is quite different from the general samples. This might explain the need for an extra component. This will be further discussed using an experiment in the obstacle environment.

It is also important to understand why this extra component leads to such an improvement in results (Figure 5.19). To show the added value of the fourth component, we analyze the T-SNE plots of the state representation corresponding to the experiments in Figure 5.19 of the underdeveloped state size three and the fully-developed and non-restricted state size ten. A two-dimensional projection using PCA would neglect the higher components we want to analyze. T-SNE plots allow us to see the clustering properties for all relevant dimensions, therefore giving a good indication of the added value of the fourth component.

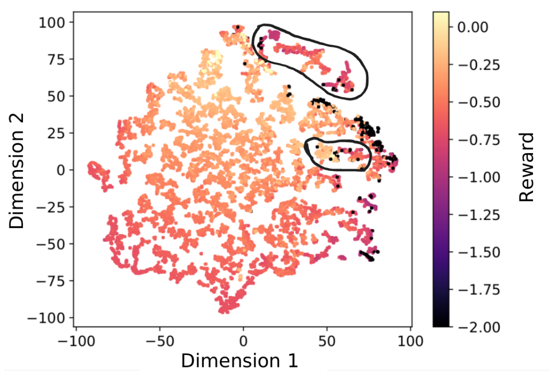

Figure 5.23: T-SNE plot of a three-dimensional state representation shows some very badly mapped regions indicated by the black circles.

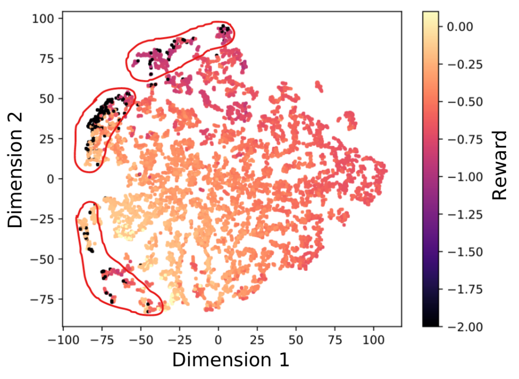

Figure 5.24: T-SNE plot of a ten-dimensional state representation can make distinction between the three walls (red circles) and shows much better clustering behavior.

By analyzing the plots in Figure 5.23 and 5.24 corresponding to state dimensions three and ten respectively, it can be seen that there are some very slight but distinct differences between the two-state representations. The overall structures of the two state representations are quite good, commenting mainly on the graduation in colors. The main difference in quality can be recognized trough some very poorly mapped regions in the left graph indicated by the black circles. Also, the clustering of the points that correspond to the target (yellow) and the walls (black) are not perfect. In the right graph, the state representation is able to do this much better and is even able to distinguish between the three separate walls as indicated by the red circles. Therefore, it is clearly visible how the extra component is able to map the observations much better. The reduced performance can mostly be explained by the badly mapped parts, as the perfect separation of the walls was also not always seen on the state representation of four and five. The badly mapped parts of the environment can be explained by the fact that a three-dimensional space is simply not enough for the network to map all observations in a correct way. This effect can also be seen using the principal components as shown in Appendix B.

In this section we have shown that the minimal amount of variables needed to encode a good state representation was four, thereby explaining the reduced performance for the two- and three-dimensional state network in the previous section. In-depth analysis has also shown that there are similarities between state predictions and certain physical properties. This emphasizes how the state representation learning method makes use of the robotic prior knowledge to encode data that is considered relevant to the task into the state representation very effectively.

5.2.3 Effect of environment size on optimal state dimension

Next, we investigate the influence of the size of the environment on the optimal state dimension. The *small four walls environment* has been scaled up, as described in Chapter 4. This new environment is referred to as the *large four wall environment*. In this experiment, the state representation network has been updated five times at a 200 episode interval and is given dimension ten,

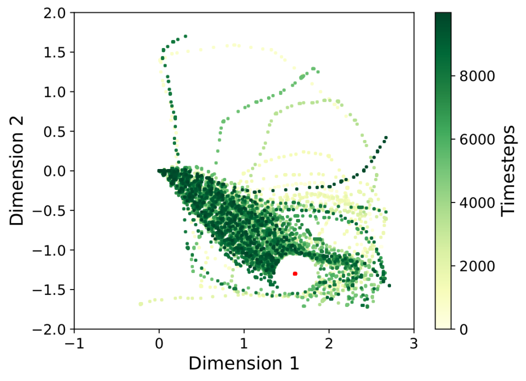

Figure 5.25: The last 10000 position samples in the *large four walls environment* indicate that the agent is able to obtain a good policy in the larger environment.

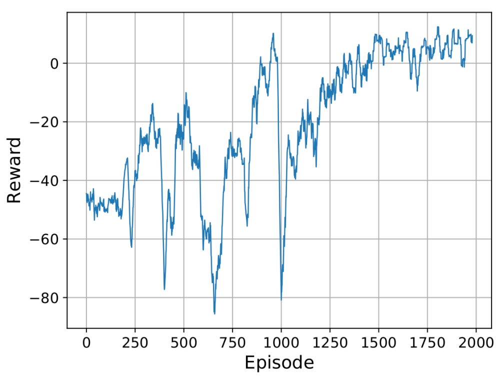

Figure 5.26: Moving average (20) episodic reward of method applied in the *large four walls environment* shows that the learning duration is effectively doubled compared to the small environment.

To demonstrate the algorithm's ability to obtain a good policy in this larger environment and to get an indication of how this policy looks like, the last 10000 position samples have been displayed in Figure 5.25. These last 10000 samples correspond to approximately the last 160 episodes as the average episode corresponds to 63 steps. At that point the method has converged and therefore gives us a good indication of the final policy. As can be seen from the graph, the agent first rotates slightly and then drives to the target in a straight line.

As can be seen from Figure 5.26, the proposed method is able to develop a good policy in 1000 episodes. After the first and second state network updates, the method is already able to improve compared to random behavior. Although, the state representation is not good enough yet to reach the target consistently. After the fourth update at the 800th episode, the method quickly improves its policy and is able to reach an average reward above 0 for the first time. This is a good indication that the agent is directed to the target consistently as the reward is averaged over 20 episodes. After the last update, it takes another 500 episodes to consistently return to this level, which can be explained by the more complex state representation that is formed. In terms of learning speed, the large environment is significantly slower than the smaller environment. A experiment in the *small four walls environment* resulted in a total training duration of around 400-500 episodes. In this environment, the training effectively takes 1000 episodes to find the target for the first time. Therefore we can say that the training takes around twice as long.

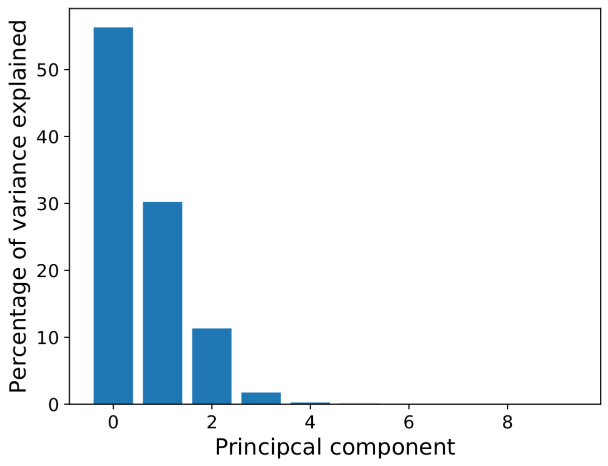

Figure 5.27: Principal component analysis of the *large four walls environment* shows that the required state dimension is not affected by the size of the environment.

To identify the required number of states needed to capture the state representation, we have performed a principal component analysis. From Figure 5.27 it can be concluded that the amount of components needed to describe the state of the agent for the *large four walls environment* is four, as the fifth component represent less that one percent of the variance. During multiple confirmation experiments in this environment, it was seen that the fifth component is almost never contributing to the variance. Since the state representation takes form of the physical properties and adapts to the complexity of the task, it can be easily explained why the bigger environment does not require extra components to encode the full state representation. This is because the scale of the environment does not affect the dimensionality of the task and the reward mapping problem and therefore does not affect the developed amount of principal components.

5.2.4 Effect of environment complexity on optimal state dimension

Next, the effect of the complexity of the environment on the optimal state dimension is investigated. In order to do so, two environments have been presented in Chapter 4. During these experiments, the obstacle avoidance method that is described in Section 4.1.1 has been employed. The reason for this is that the ϵ-greedy method by itself was limiting the exploration of the agent quite significantly. What happens is that the first part of the environment is visited many times, while the part of the environment around the corner is not reached nearly as much because of the 0.35 epsilon value. The presence of random action causes the agent to run into the wall a lot of times, even though the reinforcement learning algorithm learned how to avoid this. This results in unbalanced exploration of the environment and consequently a unbalanced learning of the state representation network in harder to reach areas.

The first experiment was conducted in the *L-shaped environment.* The experiment has been performed for a total of 2000 episodes featuring five state representation learning updates at episode 200, 400 and 600, 800 and 1000. The state size was set to ten. Figure 5.28 shows the last 10000 position samples of the agent. In this environment, 10000 samples correspond to the last 130 episodes. By analyzing the graph it can be concluded that a good policy was learned. With the exception of a couple of experiments, almost all trajectories lead directly the target. Since the inward corner of the L-shape is placed on $[0.75, -1.25]$ we can see that the policy steers the agent just past this point, meaning that the trajectory also was well optimized.

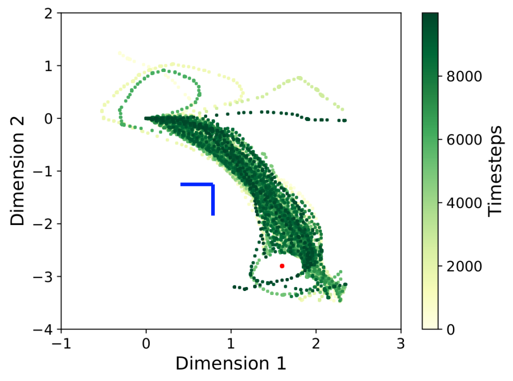

Figure 5.28: Last 10000 position samples of the experiment set in the *L-shape environment* indicates that the method is also able to work in slightly more complex environments.

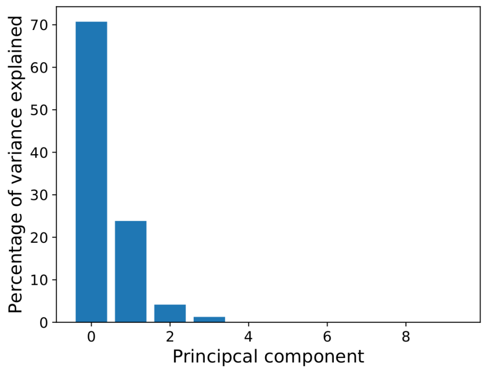

Figure 5.29: Principal component analysis of experiment set in *L-shape environment* shows that the number of required components is still four.

From Figure 5.29 it can be seen that the number of principal components that were needed to explain all variance was four. This is similar to the task of Figure 5.20 and 5.27. The logical explanation for this is similar to the experiment conducted in the *large four wall environment*. Namely that the shape of the environment does not necessarily impact the dimensionality of the task and therefore does not impact the number of required components.

For the *obstacle environment*, the experiment has been conducted for a total of 3000 episodes since the reinforcement learning algorithm needs more time to learn the more complex policy. The state representation network has been updated on episodes 200, 400, 600, 800 and 1000, similar to the *L-shape environment*. The state prediction dimension is ten-dimensional.

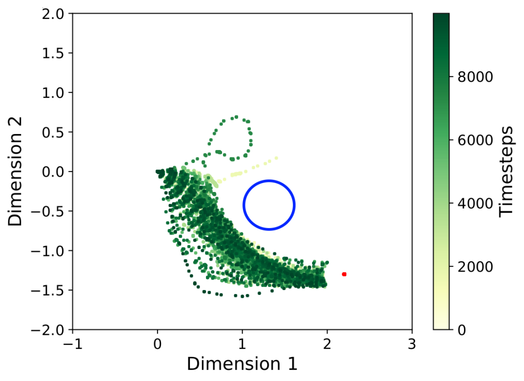

Figure 5.30: Last 10000 position samples of the experiment set in the *obstacle environment* show the the agent is nicely guided around the obstacle.

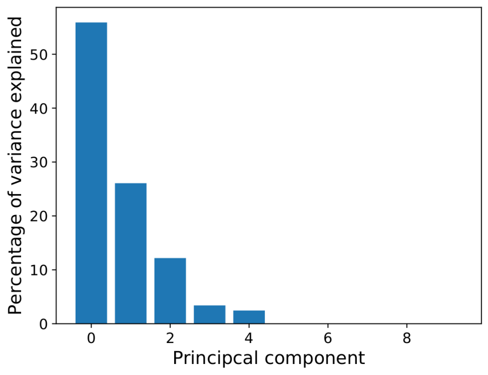

Figure 5.31: Principal component analysis in the *obstacle environment* show that more than four components are relevant in describing the state representation.

As shown in Figure 5.30, the robot manages to steer the agent around the obstacle and reach the target position. The final 10000 samples correspond to the last 150 episodes. In this environment there are two possible solutions. The agent can pass the obstacle either by going left of right. As can be seen from the position samples, the final adopted policy was optimal, passing the obstacle to the right. To guarantee that this was not incidental, multiple experiments (8) have been performed. All of these experiments resulted in this optimal behavior. In Q-learning, it is not guaranteed that the optimal solution is found. By allowing for enough exploration we can increase the chance that the optimal solution is eventually achieved.

In Figure 5.31 it shows that there are a total of five principal components that account for the total variance in the state representation in this experiment. It should be noted that the experiments in the *obstacle environment* are less consistent in terms of principal component development. Out of the eight experiments, five of them were performed using the exact same configuration as explained above. Three times out of five, five relevant components were seen. The other two experiments resulted in four and seven relevant components. Since there are many stochastic processes that can affect the behavior of the robot, the observation samples set can become quite different. While optimizing the network, there are many random factors as well (e.g. weight initialization, random sampling from experience). This can result in different state representation structures. Nevertheless, the results indicate that the behavior for the obstacle environment is different from the other experiments, as we have found only four components in the other environments. To get a more reliable indication of the number of components that is needed to capture the environment, an experiment was performed using only random actions. A rotational velocity was selected such that the part behind the obstacle was explored sufficiently. In these experiments, it was observed that five relevant principal components were seen consistently (3 experiments).

The causality prior (3.3) tries to separate states that lead to dissimilar rewards (when applying the same action), the temporal coherence prior (3.1) tries to place temporally correlated samples together, and the proportionally prior (3.6) tries to distribute the states spatially according to their relative Euclidean distance in the original environment. Besides that, the repeatability prior (3.7) tries to enforce deterministic state changes, meaning that similar reward variations (in magnitude and direction) should also lead to similar state changes (in magnitude and direction). If not possible, the states should be pushed apart. Together, these priors construct a state representation that mimics the reward space of the original environment and maintains the temporal correlations between the samples. One can understand that a sample hitting an obstacle (that corresponds to very low reward) causes a disruption of the reward space and makes the mapping problem significantly more complex. For the outer walls, this effect is present but less strong since they can be placed on the "outsides" of the state representation. This effect is even more pronounced for an obstacle in the middle of the environment. The causality prior tries to separate these outliers from the general samples because the rewards are dissimilar while the repeatability prior, proportionality prior and the temporal coherence prior try to preserve the structure of the environment - by keeping the structure intact we mean that a sample to the right and a sample to the left of the obstacle with equal Euclidean distance to the obstacle should also have equal Euclidean distance to the obstacle in the state representation. This constraint most likely leads to the fact that the samples corresponding to the obstacle being pushed away in a direction that is perpendicular to the "general" states, resulting in the fact that the variance of the data set is explained among a new component. In Appendix D it can be seen how the samples corresponding to the obstacle are separated from the general samples.

Experiments in this section have shown that the amount of components required to describe the state representation does not necessarily have to increase in more complex shapes, as long as the complexity of the mapping problem and the dimension of the task remain similar. However, the experiments in the obstacle environment have shown that an obstacle can cause a disruption in the reward space, effectively increasing the complexity of the mapping problem and therefore requiring an additional component.

5.2.5 Effect of reward function on optimal state dimension

Finally, we discuss the effect of the reward function on the optimal state dimension. For these experiments we have changed the reward function to the following:

$$r(s) = \begin{cases} 20, & \text{if reach destination} \\ -20, & \text{if crash} \\ 1 \text{ - } \mathrm{e}^{(a*\mathrm{abs}(orientation\_difference))} & \text{otherwise} \end{cases} \tag{5.1}$$

The orientation difference is defined as the difference between the orientation of the agent and the desired orientation, which is the angle between the agent and the target ranging from -3.14 to 3.14. The scaling factor α was initially set to 0.5 to match the alpha value of the distance-based reward. This resulted in reward values much higher in magnitude compared to the distance-based reward. Because this can affect performance, it was lowered to 0.13 to be comparable to the distance-based reward experiment with $\alpha = 0.5$. Another experiment was performed for the distance-based reward with $\alpha = 1.0$ to match up the orientation experiment for $\alpha = 0.5$.

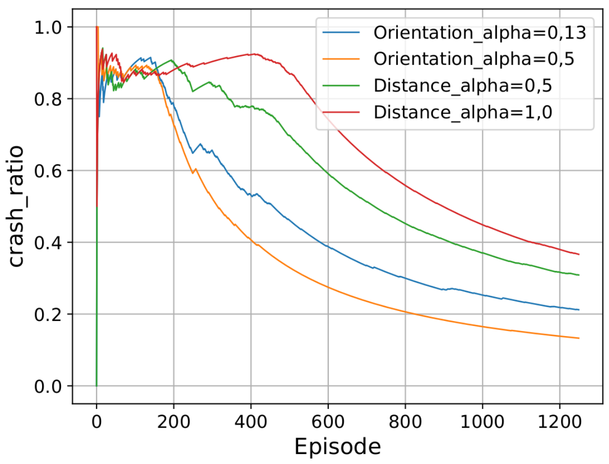

Figure 5.32: Crash ratio comparison of distance based and orientation based reward shows that the orientation based reward outperforms the distance based reward function for higher and lower alpha values.

From Figure 5.32 it can be seen that during the first 125 episodes, all experiments behave randomly since no state representation has been built up yet. After the first update, both orientation based reward functions immediately shoot down in crash ratio, meaning that they already start reaching the target consistently. The distance-based reward function is considerably in slower finding the target. The distance-based reward with $\alpha = 1,0$ takes until third state network update (375 th episode) until it starts developing a good policy. The distance-based experiment with $\alpha = 0,5$ is slightly better as it is able to find the target from the 250 th episode. Overall it can be seen that for both cases, the orientation reward outperforms the distance-based reward function quite significantly. During multiple experiments, it was seen that the orientation based reward could find the target in about 200 episodes while the distance-based reward takes on average 400 episodes. This means that training time can be effectively reduced by 50%.

Since the two reward functions behave quite differently in how they steer the agent to the target, the difference can both be attributed to the reinforcement learning problem as well as the state representation learning problem. To investigate, a similar experiment is performed using ground truth instead of the state prediction to see the effect on performance on the reinforcement learning problem alone.

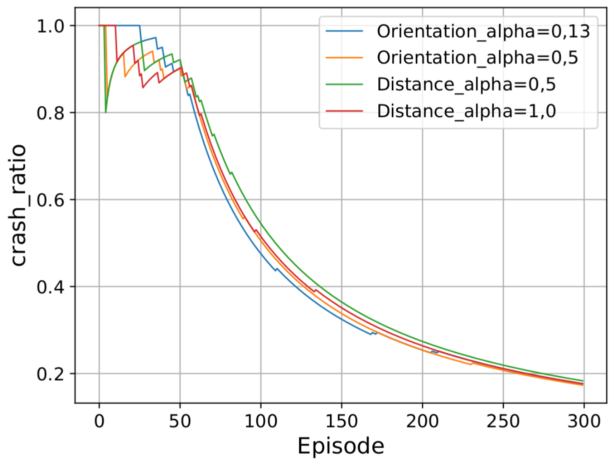

Figure 5.33: Crash ratio comparison of different reward functions using ground truth show that the reinforcement learning problem is hardly affected by the reward function for this task.

As can be seen from Figure 5.33, the results in Figure 5.32 cannot be explained by the reinforcement learning problem alone since the differences in crash ratio between the experiment are almost nothing. This means that the increase in performance can either be attributed to the quality of the state representation or the robustness of orientation based reward on input data that is not ground truth. Next, we asses the quality of the formed state representation for both methods.

Both the proportionality, causality, and repeatability prior losses exploit the rewards to build a state representation. How this information can be exploited is very depending on the reward function. A distance-based reward can help the network differentiate between states in terms of distance while the orientation based reward can help differentiate between states with different orientations. This means that the formed state representation will most definitely be affected by how the reward function is defined.

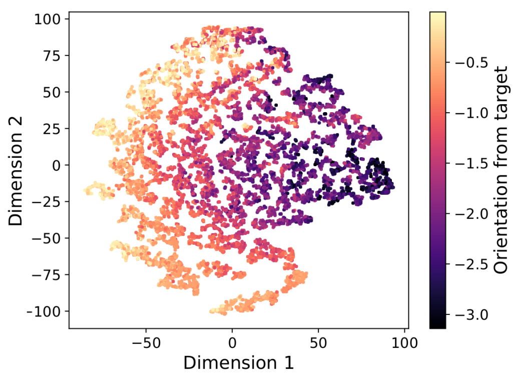

Figure 5.34: T-SNE plot of state representation (colored by orientation to the target) for distance-based rewards shows that the network is able to differentiate between different orientations but is not able to cluster them properly.

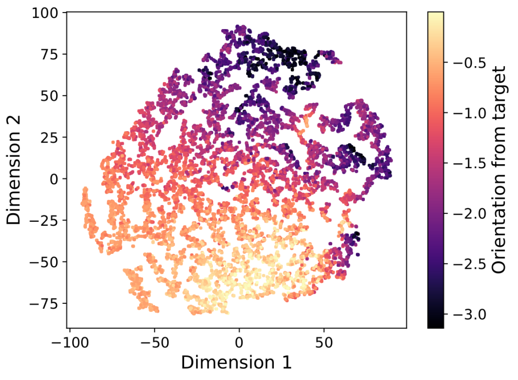

Figure 5.35: T-SNE plot of state representation (colored by orientation to the target) for orientation based reward shows that similar orientations are clustered perfectly.

As can be seen from Appendix C, the distance encoding properties of both networks are very good, as there is nice graduation from the target (most yellow samples) to the other parts of the environment. The difference in state representation is most noticeable for orientation related data. As can be seen from Figure 5.34 and 5.35, both networks have encoded the orientation difference to the target in some way since the graduations in colors are overall quite good. However, the most noticeable difference is that for the left graph, even though the color gradient is quite good, the clustering of similar orientations (relative to the target) is far worse than the clustering properties shown in the right graph. The yellow samples in the left graph are distributed along the whole edge of the graph, while for the right graph there is much better clustering of similar orientations. This is a great example of how the priors can exploit the reward function to encode useful information. The orientation difference to the target is very (if not the most) relevant information to the task. If the network is able to encode this information very well, the reinforcement learning algorithm would most likely benefit from this. This can very well explain the improved performance of Figure 5.32.

To analyze the optimal state dimension for the orientation based reward function, an experiment was conducted with a state dimension of ten and a total of five state representation network updates on a 125 episode interval.

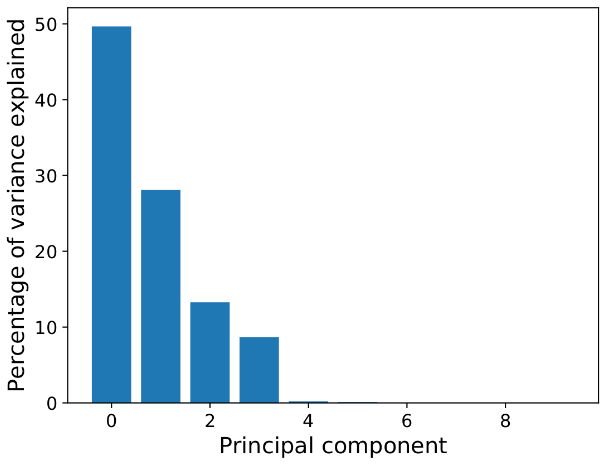

Figure 5.36: Principal component analysis of orientation based reward experiment shows no difference compared to the distance-based reward experiment.

Figure 5.36 shows that the number of components necessary to describe the state representation data set is four, similar to the distance-based reward. Looking closely, it can be seen that a fifth component is slightly present but explains less than 1% of the total variance of the data set. By analyzing the results during training it was seen that this fifth component was already present during the first state network update and was slowly decaying from that point after, meaning that this dimension most likely does not contribute to the final solution. To further analyze the structure of the state representation using an orientation based reward, a correlation analysis was performed. A new experiment was conducted in which only random actions were performed for a total of 400 episodes. The rotational velocity was doubled to mimic the more complex trajectories during policy-related actions, similar to Table 5.2. The state representation was trained for five epochs.

Table 5.3: Correlation coefficients of principal components and physical properties using orientation based reward and random actions.

	X-position	Y-position	Distance to target	Orientation
Principal component 1	0.80	-0.01	-0.35	-0.08
Principal component 2	0.17	0.72	0.55	0.67
Principal component 3	0.06	- 0.19	-0.16	-0.52
Principal component 4	-0.36	0.02	0.23	-0.04

In Table 5.3, it can be noted that the structure of the principal components is very similar to that of the distance-based reward in Table 5.1. The first component translates to the x-position, the second component translates to the y-position and the third component mostly corresponds to the orientation. It is very interesting to see that even though the reward function is much more aimed toward the orientation of the agent, the notion of position is captured almost evenly well. Also, it is remarkable that the structure of the correlation coefficient is very similar to that of the distance-based reward even-though it was seen that the reward function can have considerable effects on the state representation (Figure 5.34 and 5.35).

In this section it was shown that the orientation reward outperforms the distance-based reward function. This difference cannot be attributed by reinforcement learning by itself considering that the experiment using ground truth were all very similar (Figure 5.33). It was shown how the reward function can affect the structure of the state representation. The orientation-based reward was much better in clustering points with a similar orientation, which can be a huge advantage for the reinforcement learning problem. To further analyze the difference in performance it should be investigated how important the orientation is compared to position for solving the task. This can be done using selective ground truth data (only orientation vs only position). Furthermore, it should be investigated whether the difference in performance is not caused by the fact that the orientation based reward is advantageous for the reinforcement learning algorithms in case of a non-optimal state representations. This can be done by transfer learning both state representations (distance-based and orientation-based) to a task that uses the same reward function.

5.3 Results corresponding with RQ2

In this section we will feature results regarding the second research questions that aim to investigate more realistic use-cases.

5.3.1 Generalization properties

To investigate generalization properties of the state representation learning method, several experiments were performed in the *large four walls environment*. First, the agent was trained to navigate to the bottom right position $[1.6, -1.3]$. This was done for a total of 1700 episodes, updating the state representation network a total of five times on a 200 episode interval with a ten-dimensional state dimension. The last 700 episodes were used to form a good policy, but are irrelevant to the transfer learning properties. After that, the state representation network weights trained during this experiment were transferred to five different tasks. These tasks involved reaching five different target positions, with four being in each of the corners of the environment and one being in directly in front of the robot as shown in Section 4.3. Each of these tasks received the training weights from the pre-trained state representation learning network. This network was kept fixed and each of the experiments proceeded to learn a policy based on the pre-trained state representation learning network for a total of 1250 episodes.

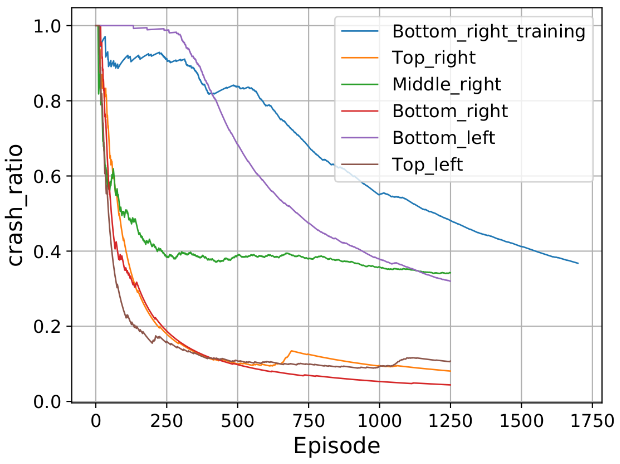

Figure 5.37: Crash ratios of the transfer learning task experiments show that transfer learning can be very successful for specific tasks but can also result in poor performance for other tasks.

By looking at Figure 5.37 it can be seen that the pre-training experiment converges slowest. This is caused by the fact that a large part of the experiment was devoted to training the state representation. This also shows why it can be extremely beneficial to use transfer learning from task to task. Now commenting on the transfer learning tasks, we can see significant differences between the different target locations. The task corresponding to the target in the bottom right performed best. This is understandable since the task is equal to the training situation. The top right and the top left target performed almost equally well, although there are some inconsistencies during training as can be seen from the spikes in the crash ratio. The target in the middle right and the bottom left performed very poorly and were not able to solve the task consistently. During training, there is a reasonable amount of exploration. However, when training on one single target it is not guaranteed that all parts of the environment are explored. This is convenient because it reduces training time, but can result in poor mapping in these under-explored regions when performing transfer learning. In Appendix E it is explained how under-exploration of certain regions can explain the difference in results. In here it is shown that the experiments corresponding to the result that performs worst also receive very bad state predictions. This is caused by the fact that the observations corresponding to reaching that target are not trained-on well enough.

Our results have shown that a network trained for a specific target can be successfully transferred onto tasks with different targets. This means that the state network is able to produce a task general state representation, which is one of the necessities for an agent that needs to be able to navigate to multiple target positions. This also shows that the introduction of the state network can separate the task from the environment and therefore reduce training time by transferring the state network to different tasks, which would not be possible using *end-to-end reinforcement learning*.

However, this only holds when the targets are placed in a region that is trained on well enough. It was also shown that less training in certain regions can result in poor performance. This means that when multiple target locations are required, it is favorable to train the agent on all possible trajectories. In this case, random exploration could be preferred at the cost of longer training duration. Another option is to train the state representation network and the reinforcement learning network simultaneously on multiple target locations. The problem is that every time a target location changes, the reward space of the environment also changes. The state network should be able to make a distinction between the different reward allocations with respect to the environment. This will be further discussed in the recommendations section.

5.3.2 Realistic environment

In this section, it is aimed to show how well the state representation performs in more realistic environments. In Section 4.3 it was shown how a real environment was replicated using pictures (Figure 4.9). The results in this section are based on experiments in this environment. To be able to address whether the added features have a positive or negative effect on performance, a similar experiment was performed using colored walls. Both experiments were given a state dimension of ten and were updated a total of five times on a 200 episode interval. Other than the features on the walls, the experiments are identical.

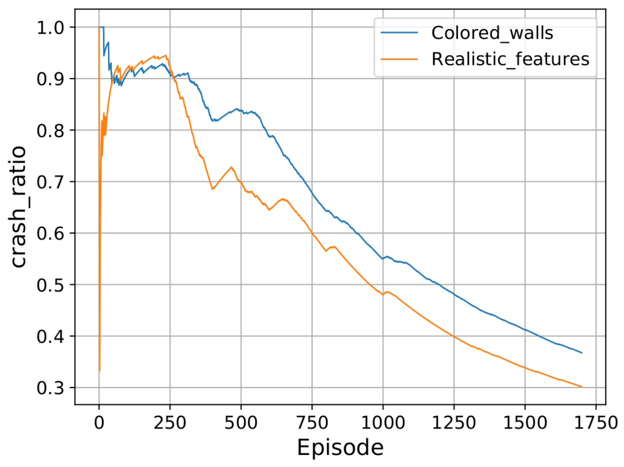

Figure 5.38: Crash ratio of experiment in the realistic environment shows that the more realistic features can even improve training speed.

First of all, Figure 5.38 shows that the agent is able to solve the task in the realistic environment. Comparing the two methods, it can be claimed that agent performs even better in a realistic environment. This difference is most noticeable during the first 700 episodes. Afterward, both methods are able to reach the target consistently. It is noticeable is that the agent in the realistic environment suffered significantly more from the state network updates at the 400th, 600th, 800th and the 1000th episode. This is explained by the fact that the more realistic features are far more complex and their importance can change over time.

It can be concluded that the state representation learning network is also able to extract useful information out of a more realistic environment. The colored walls are a significant simplification, making it very easy for the environment to find its orientation (which we think the camera data is mostly used for in these environments). In this case, the network needs to extract more complex features that are present on the walls that can yield information regarding its state. The results show that the prior losses are indeed able to shape the convolutional filters such that the relevant features are taken out of the environment in order to estimate the state. It was even shown that these features can decrease training duration.

In terms of shape, the environment is still very simple. More experiments should be performed to test how well the method will work in more complex shapes. Also, it should be noted that there is no emphasis on the structure of the state representation learning network. When using more complex environments, it is very important to have more focus on the convolutional layers since these are key to extract the relevant features.

5.3.3 Transfer learning to real-world situation

In this section, we will discuss our findings regarding transfer learning. The idea of transfer learning is to transfer knowledge from one situation to another. In this case we investigate whether it is possible to transfer knowledge obtained during simulation to the real robot. Transfer learning from simulation to the real setup is a very interesting proposal in order to cut expenses and training time. In this section we present our results regarding transfer learning and describe our trial and error process.

The first step was to obtain the simulation results. In Chapter 4 it has been described how the experimental setup with boxes has been recreated in simulation. The pre-training has been performed on this simulated environment for a total of 1250 episodes, obtaining the network weights for both the state representation learning network and the reinforcement learning network. The state representation network is updated only once at the 250 th episode. The reasoning behind this is to prevent overfitting to the training situation. To verify the learned networks and to be able to compare performance, we performed an evaluation round where the random actions of the agent were set to none ($\epsilon = 0.0$) and the experiment was set to perform the task 100 times. The agent successfully performed the task, reaching the target 100/100 times. The trajectories during this evaluation round are presented in Figure 5.39.

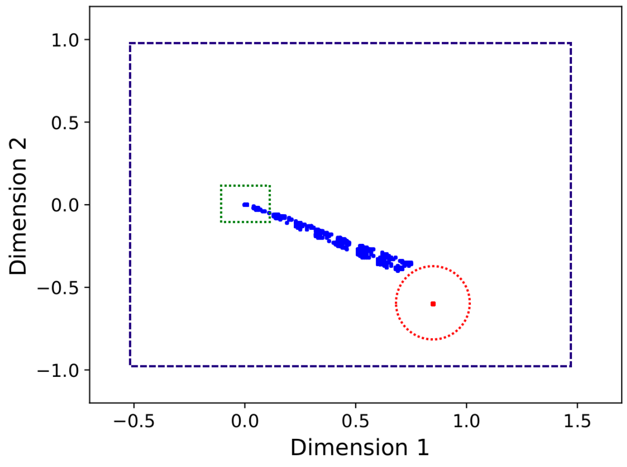

Figure 5.39: Evaluation round of pre-trained network in simulation shows that the networks are trained correctly.

After confirming the state prediction network and the policy, the weights of the network have been transferred to the real robot. The following results correspond to the first set of tests on the real system. The experiment was set to do ten evaluation rounds, meaning that epsilon was set to 0. The results of this evaluation set is displayed in Figure 5.40.

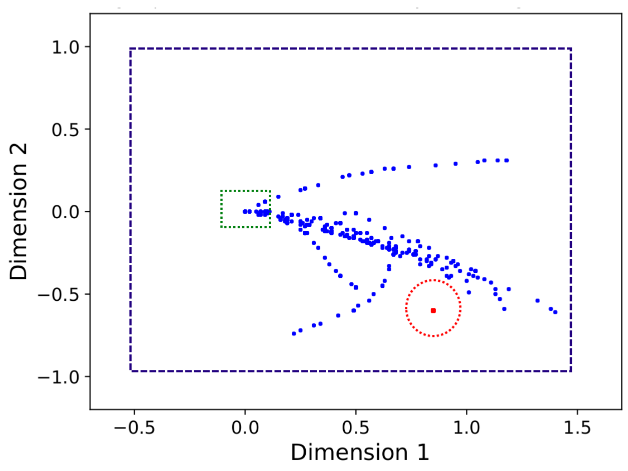

Figure 5.40: First evaluation round of pre-trained network on real system shows that the real robot is not able to reach the target.

During this evaluation round, the agent did not manage to reach the target. The majority of the actions consisted of the agent rotating left and right on its place. This caused the agent to exceed the step limit per episode, hence the number of trajectories that do not result in the agent crashing or reaching the target. This rotating behavior did not appear to be random since the focus of the agent was towards the lower right corner where the target is located.

To improve performance, the setup was moved to a location with a fully white background. In the initial experiments, the background had several distractions. In the next set of experiments, we show that we are able to successfully employ transfer learning from simulation to reality. In this experiment, the only difference with the previous result is the background visible to the agent. The agent was set to run for a total of eleven evaluation rounds. The corresponding trajectories are shown in Figure 5.41.

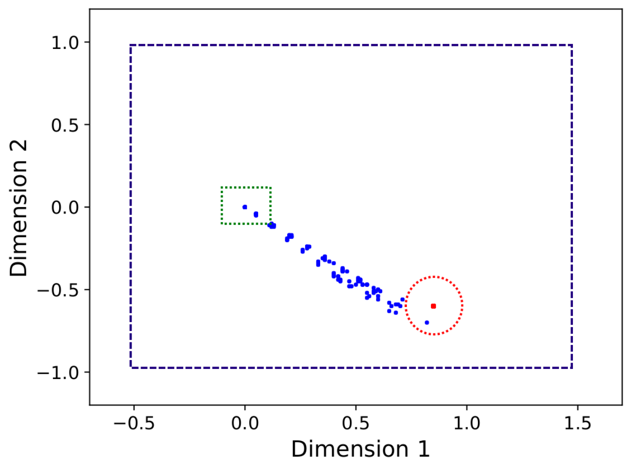

Figure 5.41: Second evaluation round of pre-trained network shows that the real robot is reaching the target after the distractions are gone.

The agent was able to reach the target position 11/11 times. The oscillating behavior was totally gone and the agent was guided directly to the target. To get an indication on the scale of the experiments, we show three action shots during one of the successful runs below.

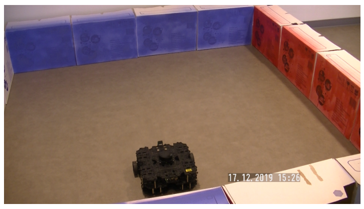

Figure 5.42: Turtlebot on initial position.

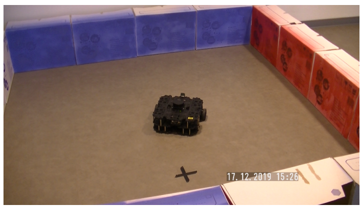

Figure 5.43: Turtlebot moving towards target.

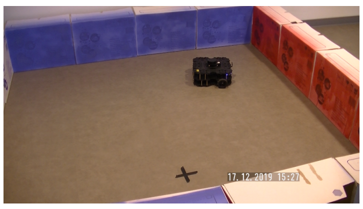

Figure 5.44: Turtlebot reaching target position.

One week later the experiments where repeated. Interestingly enough, the agent was not able to achieve the same performance. The agent was unable to reach the target. The most apparent changed factor was lighting. Therefore, the first step was to obtain a better understanding of the effect of the lighting. In the testing room, two sources of light were present: a light bulb and a window. To investigate the effect of the lighting the light bulb was turned off, leaving only the natural light. During these experiments, the agent behaved very differently and far from desirable, running into the outer edges almost all the time.

The initial reason to train the state representation network only once was to not over-fit to the simulation. However, the network can also be under-developed. A new experiment was performed on the same day using a state network that was updated three times. This time, similar performance to Figure 5.41 was achieved. To obtain more insight regarding lighting, we performed multiple experiments, starting with the light bulb switched off. After a while, the lights were turned back on. These experiments led to some interesting results. While the light was off, the agent could not correctly predict its position and was driving in circles. However, the agent did not crash. As soon as the light turned back on, the agent immediately turned towards the target position and drove in a straight line towards the target. In Figure 5.45 we show how the agent turns in a circle while the light is off (purple) and immediately reaches the target position when the light is turned back on (blue).

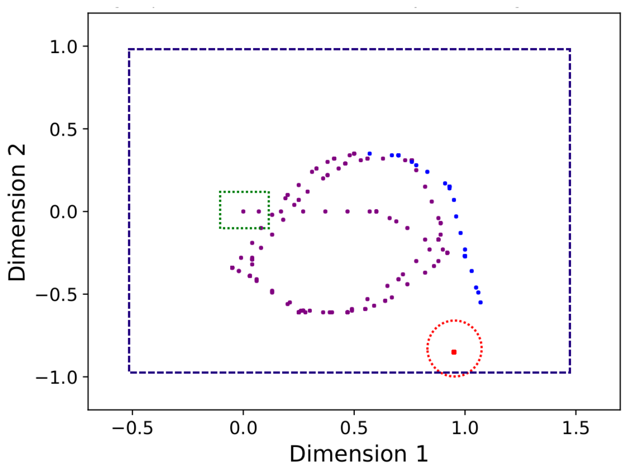

Figure 5.45: Evaluation round of pre-trained network on real system with lights turned of (purple) and lights turned on (blue).

These experiments have shown the possibilities of transfer learning simulation-based data to a real system. We believe the main reason for the success of transfer learning was the use of the state representation network. The encoded data is far less affected by noise, disturbances, and discrepancies between simulation and reality, which benefits the reinforcement learning algorithm. This still means that the state representation network should be robust against factors.

Although, it simplifies the problem. Another reason for success is the use of highly down-scaled pictures and the use of laser data. This is easier to match in simulation compared to very high-resolution images. One important consideration is lighting. We have shown how much impact the lighting configuration can have. In the next chapter, we will propose a solution to this problem.

5.4 Performance analysis

In this section, the relevance of state representation learning will be displayed. Besides that, it will be shown how the method compares to the ideal situation. The experiments are located in the *small four walls environment* and are run for a total of 1250 episodes. This section will include the results of three different methods:

The first method is reinforcement learning by itself based on ground truth. The agent is provided with the x- and y-position of the agent and the sine and cosine component of its orientation. This is considered the most optimal input data and will be used as an ideal-scenario reference.

The second method is a combination of reinforcement learning and state representation learning. In this case, the state network is updated three times with an interval of 125 episodes. The state dimension is set to five.

The third method is again reinforcement learning by itself. In this case, the inputs are not the ground truth state, but the inputs supplied from the camera and the lidar. The neural network used in the state representation learning method has been combined with the reinforcement learning network to ensure a fair comparison. Both methods have convolutional layers to extract useful features and both methods have exactly the same amount of parameters. The only difference is that the first part of the network is not updated by the priors anymore, but is updated from the reinforcement learning loss.

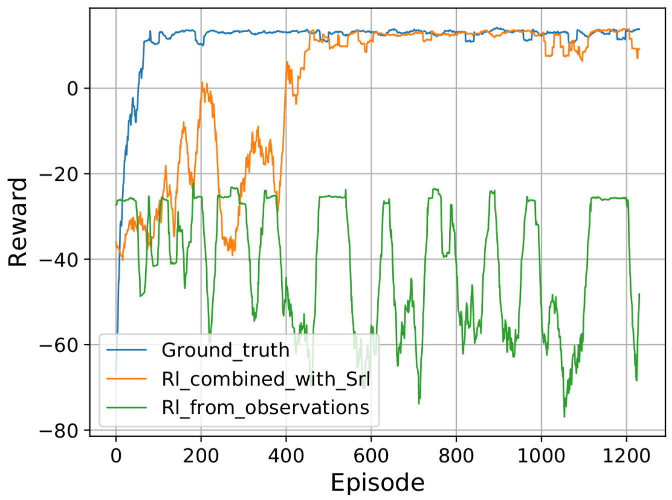

Figure 5.46: Moving average reward (20) comparison of ground truth, only reinforcement learning, and state representation learning combined with reinforcement learning experiments.

As can be seen from Figure 5.46, the ground truth experiment results in a good policy in under 100 episodes. Ground truth input is considered to be the ideal scenario and is provided as a baseline in order to assess the performance of the other methods. The state representation learning combined with reinforcement learning experiment takes approximately 400 episodes in order to find the target. The reinforcement learning method that learns directly from the observation is not able to construct a good policy. Neither does it have any implication of any progress. Even though the two methods have exactly the same neural network structure, the reinforcement learning method directly from observation is simply not able to connect the high dimensional input to the low informative reward. This is a typical demonstration of the curse of dimensionality in reinforcement learning applications in robotics. The state representation network resolves this by extracting relevant information by using robotic prior knowledge. We have shown the added value of state representation learning by effectively solving the dimensionality problem of reinforcement learning by extracting relevant information from high dimensional input data using prior robotic knowledge. This allowed the state representation learning method to find a policy in approximately 400 episodes compared to 100 episodes when using ground truth. The reinforcement learning method directly from observations did not achieve to find a policy leading to the target.

5.5 Critical assessment

This section features a critical assessment regarding the method proposed in this thesis. We distinguish several strengths and weaknesses below.

Strengths

- *Enables reinforcement learning for tasks that require high dimensional sensor inputs*: From Figure 5.46 it was seen that reinforcement learning by itself is not able to solve the task. This is due to the curse of dimensionality. As aforementioned, state representation learning can solve this problem. This enables reinforcement learning in a much wider range of tasks in robotics.

- *Only captures relevant data to the task*: The proposed state representation learning method only captures data relevant to the task, ensuring that the reinforcement learning network does not need to further extract relevant information from the state representation. This is enforced by making use of the rewards and actions to shape the state representation

- *Can make use of features in realistic environments*: The proposed approach has proven to be useful in realistic environments. In literature, the applicability of the method was restricted to artificial environments. By making use of (multiple) convolutional networks to extract high-level features it was shown that the proposed approach is not restricted to highly simplified environments.

- *State prediction separated from reinforcement learning task*: A great advantage of the proposed algorithms is that the state encoding stage is separated from the reinforcement learning problem. First of all, this can greatly reduce training time but it also allows for transfer learning between tasks. This means that the state network weights can be transferred to different tasks in the same environment. In this way, only the reinforcement learning network has to be retrained.

- *Transfer learning from simulation to real system*: In this thesis, it was shown how the network weights generated in a virtual environment can be successfully transferred to the real system. This was done without any adaptations to parameters or additional training. Transfer learning from simulation to reality can be considered as a significant advantage since gathering samples on a robotic system is often very challenging and expensive. Simulations do also enable to obtain experience much faster since the real-time factor can be increased. Transfer learning can be considered as a key element to reinforcement learning in robotics.

- *Non-hardware dependent*: Another advantage of the robotic priors is that they are independent of the observations. This means that theoretically any combination of sensor modalities can be used as long as the observations can uniquely describe all relevant states. It is desirable that the observation space is smooth (no sudden jumps in neighboring states) since it simplifies the mapping problem. However, as long as the input modalities are properly handled (correct layers e.g. convolutional networks) and they can uniquely describe all states, any sensor configuration can be used that suits most to the task.

Weaknesses

- *Sensitivity*: As we have seen from the experiments corresponding to the real system, lighting conditions can have significant impact on results. Also, camera data is much harder to match in simulation compared to e.g. laser data. However, as we just mentioned, the state representation learning method itself is not restricted to specific sensor modalities. Therefore, a more suitable sensor configuration can be selected according to the requirements.
- *Discrete control*: One of the drawbacks of the selected reinforcement learning approach is that it uses a discrete set of actions. This can result in less than optimal trajectories and the overall movement pattern does not look smooth. Also, it can become a limiting factor in narrow environments when more complex navigation patterns are required.
- *Repeatability*: In order to ensure that the results are not incidental, multiple runs have been performed for each experiment. Initially, the variance among experiments was quite high. To minimize this, the action method was changed from setpoint-error to duration. Also, the first 150 episodes were set to follow purely random actions rather than follow a randomly initialized policy. This ensured a more even exploration of the environment among the experiments. Overall, the variance among the experiment has been reduced significantly but must still be considered when performing experiments.
- *Need for ground truth data during training*: During the experiments, we mostly used a reward function based on the distance from the target. During evaluation, the rewards are obsolete and the proposed approach is able to solve the task solely from lidar and camera images alone. However, the training stage still requires the distance from the target to be known. This can be considered a disadvantage. Although, the distance can be easily supplied by for example a beacon that uses time of flight to determine the distance (one sender GPS system equivalent). In simulation, this problem does not exist. This means that transfer learning would eliminate this problem too.
- *Random exploration*: The exploration using the ϵ-greedy policy is random. This can be a significant disadvantage in more complex environments because harder to reach areas will almost never be visited.

6 Conclusions and recommendations

Reinforcement learning is an upcoming field within robotics. It is able to solve complex problems that do not appear to have obvious programming solutions. In this thesis, we have discussed that reinforcement learning by itself can be very effective. However, tasks that require high dimensional input data often are problematic for standard reinforcement learning algorithms. In literature, state representation learning has been presented as a solution to this problem. In this thesis, we have further investigated and improved upon this method.

In order to contribute to the research regarding state representation learning, two main research branches have been proposed. One of these branches was to find the optimal state representation respective to the task. The research question accompanying this line of research was as follows:

What is the optimal output dimension of the state representation learning network for the studied reinforcement learning problem?

We first compared the performance of a simple navigation task using several selections for the state dimension (Figure 5.19). We have seen that the experiments with a too-small state size performed significantly worse and were far less consistent. The experiments with a higher state size began suffering from the curse of dimensionality and therefore also performed worse. Theoretically, the computational costs for these experiments were slightly higher. The difference in terms of experiment duration was almost not noticeable.

To investigate the influence of the environment on the required state representation, we presented several different environments. In the *large four walls environment* and the *L-shape* environment, the required amount of components to encode the state representation was not found to be affected (Figure 5.27 and 5.29). The reason for this is that the dimension of the task and the complexity of the mapping problem does not change. However, the obstacle environment strongly indicates that at least one extra principal component was required. This was found to be due to the disruption in the reward space that makes the mapping problem significantly more complex. These results show us that the complexity of the reward space should be taken into account when selecting the state dimension. Furthermore, these experiments have shown that the studied state representation learning method can be applied to more complex problems.

To investigate the influence of the reward function we have proposed a reward function based on the orientation of the agent. The most interesting observation was that the reward function can really help structure the state representation. In the case of the orientation reward, the clustering of similar orientation samples was much better. The training duration of the orientation based reward experiment was reduced by 50% compared to the distance-based reward function.

Further analysis regarding the state representation turned out that the state network is able to encode physical properties into its state representation. This is very interesting since these physical properties have not been introduced into the loss function.

According to the results regarding the first research question, we conclude the following: the number of required components depends both on the dimension of the task (amount of different movement directions) and the complexity of the mapping problem. The complexity of the mapping problem is mostly defined by the reward space. The optimal state dimension for the tasks corresponding to this thesis is considered to be between 5 and 10. This allows for sufficient freedom during development of the state representation and ensures decent compression ratio without losing relevant information. By keeping it relatively small we can also combat overfitting.

Next, we reflect on the second research question:

To what extent can the combination of reinforcement learning and state representation learning be used in real-world scenarios?

Taking into account this research question, we have proposed three types of experiments. First, we have investigated how well the method generalizes against multiple target locations. This is important since more realistic use-scenarios will most likely feature more than one target location. As a first step in this research, we have demonstrated how the state representation can be transferred through multiple tasks. By showing that the agent can successfully transfer the state networks to different targets, we have shown that the state representation can be task general and that transferring the state representation onto different tasks can possibly reduce training duration. However, it should be noted that it is only possible for locations that the state network has trained sufficiently on.

Next we have shown how the method performs in a realistic environment. This simulated environment is replicated from a real environment, meaning that a lot more features are present compared to a homogeneously colored wall. Results have shown that the agent was able to find a policy even faster compared to an environment with colored walls because it provides a lot more features for the convolutional network to extract. Most literature focuses mainly on very simplistic and unrealistic environments. Our results have shown that the method can also be applied in more industrial scenarios.

Finally, we have shown how experience obtained in simulation can be transferred to a real robot. After getting over some minor problems we have shown that we were able to achieve 100% success ratio during the experiments on the real turtlebot by transferring networks generated in simulation directly to the real robot (Figure 5.41). We believe transfer learning has been made possible by the fact that the state representation only encodes important information. This encoded form of information is far less dependent on the differences between simulation and reality. This still requires the state representation learning method to be robust against these differences. However, this problem is easier to solve compared to *end-to-end reinforcement learning.* To our understanding, there is no literature that presents transfer learning from simulation to reality for the studied case. Therefore, we consider the results obtained in this thesis to be very promising for further research and also for potential implementation in more industry-oriented applications. However, there are a lot more challenges that need to be addressed, some of these will be addressed in the next section.

6.1 Suggestions for future work

Below, several directions for future research are presented that relate to state representation learning in general.

Force similarities in subsequent state representations
In this thesis, it was mentioned how an update of the state representation learning network would result in a temporary decrease in reward, imposed by the fact that the reinforcement learning network has to adapt to the altered/shifted state predictions. During the proposed experiments, the effect was not significant since the tasks were relatively simple. This can be much more of a problem when the trajectory is very long. The agent finds itself most of the time adapting to already established parts of the environment before it can explore the final part of the trajectory. The easiest solution to this is to normalize the state representation before entering the reinforcement learning network. However, only normalizing the data would not restrict the network in its rotation. A solution to this would be to perform PCA analysis of the state representation (without reducing the dimension) such that the states always stay aligned to the most important principal components (and physical properties). This does assume that the importance is not changed during the experiment. Another idea is to enforce similarity trough a loss function. For example, penalizing the network to map similar observations differently compared to a previous version of the state representation.

Further investigation of physical properties
Using the correlation coefficient it was shown how the states encode information related to the physical properties. However, the correlation coefficient does only measure linear relationships between the variables. To further investigate how relevant properties are encoded in the network, a simple supervised feed-forward network can be introduced that tries to map the predicted state representation into these relevant properties. The accuracy of the mapping can be used to determine the quality of the state representation. This can be very useful during development stage when trying to optimize parameters or when developing new priors. In Morik et al. [34], this idea is referred to as a "validation network" where it is used to asses performance and to be able to visualize the state predictions more intuitively.

State representation learning in partially observable environments
In this research, all of the proposed environments were quite simple and share the following property: the state of the agent can be predicted using one observation only, meaning that each observation has the Markov property. This is a significant simplification of the state representation learning problem. This simplification is almost unavoidable during early steps in research. However, more realistic use cases do not always allow for Markov observations. As shown in Morik et al. [34], a long short term memory (LSTM) network can be used to solve the problem. We believe adding memory to the state network is one of the necessities for state representation learning to become useful in more realistic use cases.

Decouple state network from rewards
By using robotic priors, we rely on the reward function to form the state representation. The reward function is a very informative and conveniently available data source. As aforementioned, the downside of this is that whenever the target changes, the reward space changes with it. This imposes problems when trying to learn multiple target locations. In this thesis, we made the assumption that the distance to the target location was available. If we make the assumption that we can obtain the distance to a target, we can also find the distance to a certain reference point in the environment. We can use the distance to the landmark as opposed to the reward function for the state network. This source of information is independent of the target location, allowing for a task-general state representation that can be trained on multiple targets.

Prioritized replay
During training, the agent explores many trajectories that do lead to the target. All samples are

favored equally during experience replay. This means that the agent tries to fit many samples that are not relevant during the operation stage. This takes extra time but also decreases the prediction quality of the more important samples. In Schaul et al. [24] it is shown how prioritized replay can extract relevant transitions from a set of highly redundant failure cases by favoring transitions with a large TD-error. This reduces the amount of training on samples that are already well established. In a similar way, the state representation learning network can prioritize certain samples. For example, we can prioritize samples that have positive Q-value variations, meaning that these transitions lie on the favored trajectory. We should take the variations in Q-values rather than Q-values directly since we do not want to favor samples close to the target specifically. We should take Q-values rather than rewards since taking the direction that leads to the highest reward does not always guarantee to reach the target.

Next, we discuss recommendations that are specific to the use-case presented in this thesis.

Addition of sensor modalities
As shown in this thesis, the camera and lidar can be used complementary. This observation is most likely not limited to the combination of these sensors only. The camera and lidar were chosen because these types of sensors are very common on mobile robots designed for navigational tasks. Almost in all cases, odometry data is available as well. Odometry tries to estimate the robot's state using wheel encoders and a kinematic model. Odometry can supply very precise measurements, however, this source of data suffers from drift because of inaccuracies in the kinematic model and other physical factors (e.g. slip) that accumulate during the experiments. Camera and laser data are a less precise source of information but do not suffer from drift since they provide information relative to objects in the environment. By adding odometry data to the state representation network we could potentially benefit from the advantages of both.

Domain randomization
We have shown the importance of lighting during the tests on the real robot. Lighting is one of the main problems in transfer learning together with disturbances. To combat these, domain randomization is proposed as described in Tobin et al. [35]. This research employs transfer learning on a visually guided robotic arm using domain randomization. Domain randomization is the idea to randomize objects, colors, physical properties, backgrounds, etc. to be able to train the network in a large variety of simulated environments to ensure that the real scenario is included in the training set.

Further investigation of transfer learning possibilities
During this thesis, it was shown how transfer learning can be used to transfer knowledge from simulation to reality. As we have addressed before, transfer learning tackles the main problems of reinforcement learning in robotics as it can reduce operation costs immensely and speed up the learning process. We believe that state representation learning can be one of the key factors for reinforcement learning in robotics. The environment used in this thesis was rather simple and easy to mimic in simulation. It would be very interesting to see if the results can be reproduced in more complex environments. This would require additional research on how an environment can be recreated in simulation as realistically as possible in an efficient way. Furthermore, it would raise the question of what sensor modalities would be most fit for the tasks. For example, a lidar (given enough features) combined with odometry data could be sufficient for the task. These sensors modalities are significantly easier to match in simulation and reality compared to a 3D environment observed through a camera.

A Effect of rotational velocity on the distribution of samples

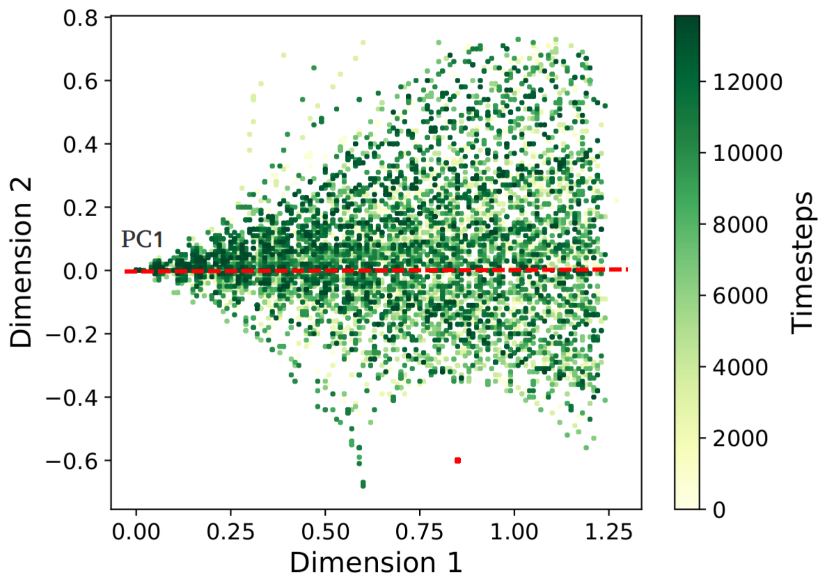

Figure A.1: Fewer relative rotations means that most of the position samples are distributed in x-direction.

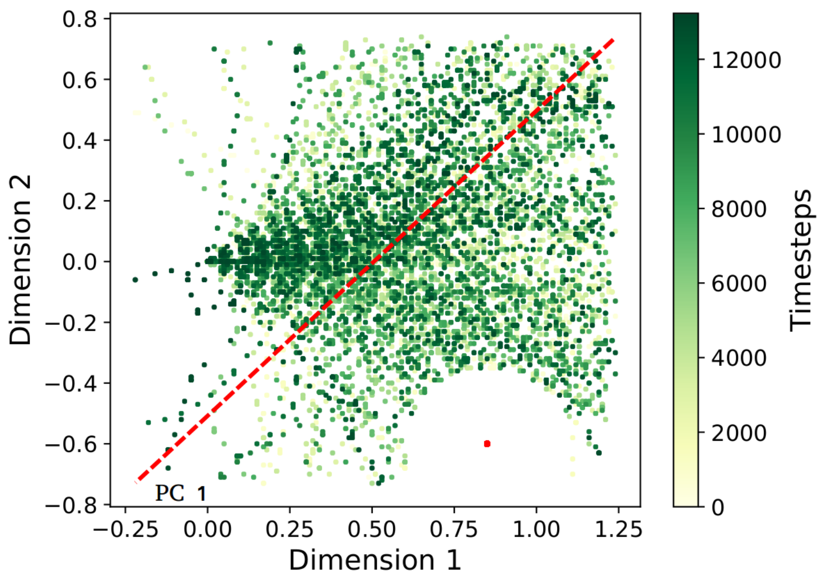

Figure A.2: More relative rotation distributes the samples more evenly, meaning that most variance is seen diagonally.

B Effect of the state dimension on state representation quality

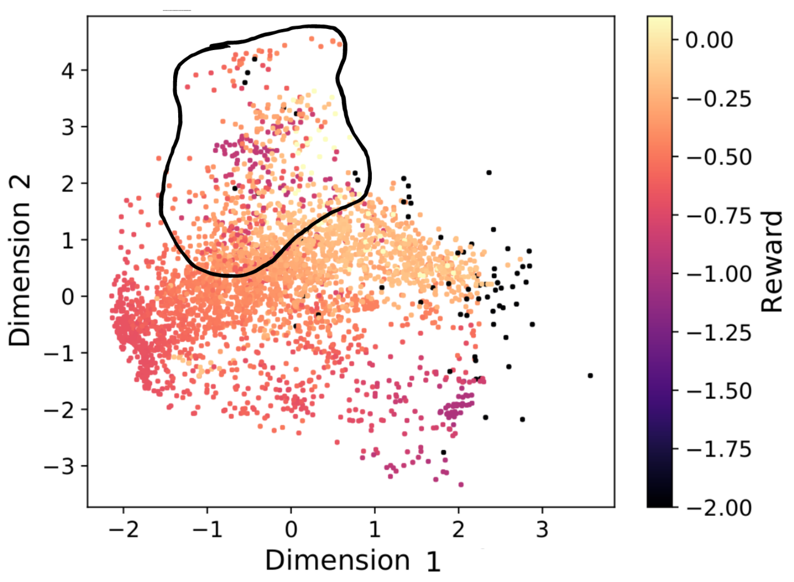

Figure B.1: First two principal components of a three-dimensional state representation shows regions that are very poorly mapped (indicated by black outline)

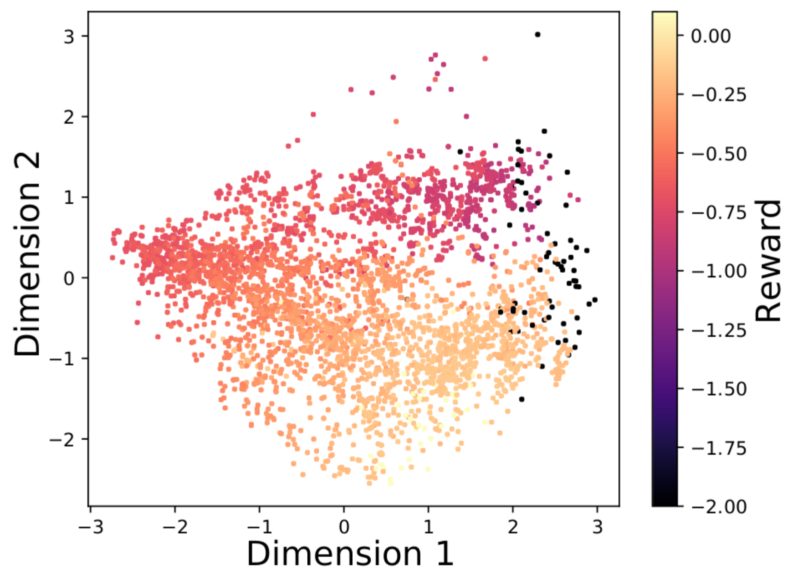

Figure B.2: First two principal components of a ten-dimensional state representation shows very good reward gradient and all samples mapped well.

C Distance encoding behavior for different reward functions

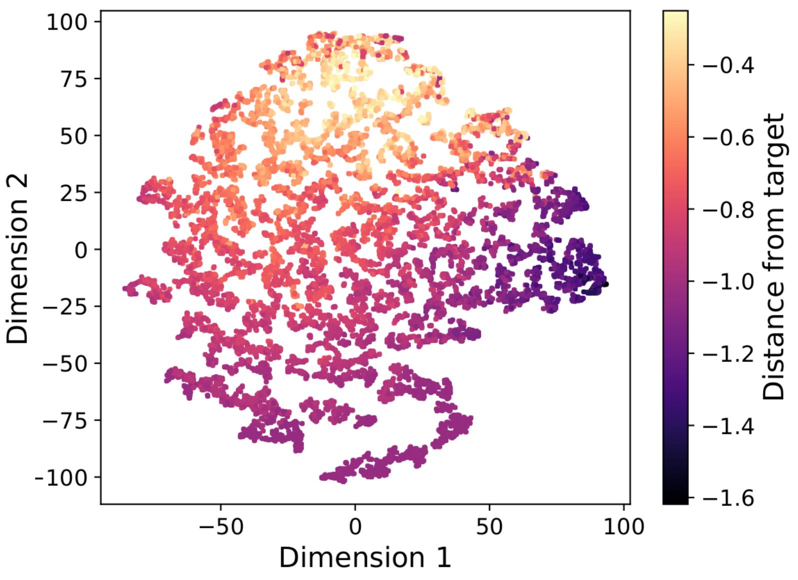

Figure C.1: T-SNE plot of state representation (colored by distance to the target) for distance based reward shows that distance is captured very well as the samples are distributed according to the distance to the target.

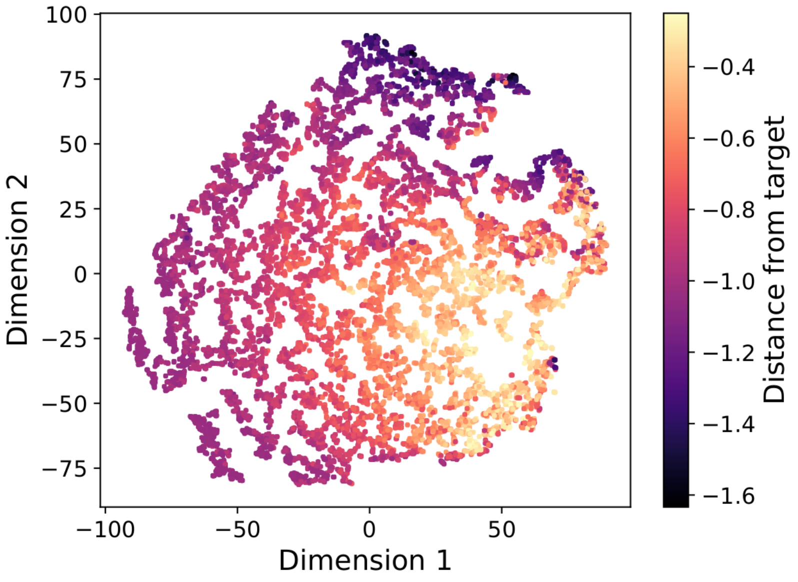

Figure C.2: T-SNE plot of state representation (colored by distance to the target) for orientation based reward shows very decent distance-encoding, even tough distance related data was no supplied.

D Separation of samples with dissimilar rewards

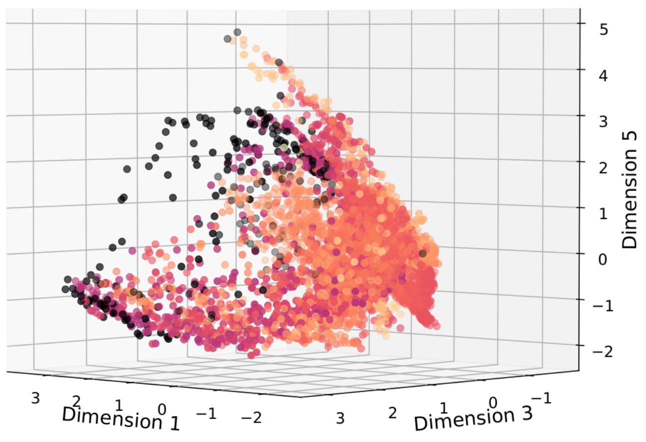

Figure D.1: A three-dimensional plot of the first, third, and the fifth component show how the obstacle related samples (black) are separated from the general samples (orange/red). This is caused by their dissimilarity in reward compared to the general samples, making the mapping problem more complex.

E State predictions for transfer learning tasks

This appendix features the first two principal components of the state predictions of the transfer learning tasks. These state predictions are based on the observations from the different transfer learning tasks, mapped by the state network that is mainly trained on the bottom right corner. These different target locations feature different types of observations (also observations that the network was not trained on a lot), meaning that some are very hard to map. By analyzing these plots we can explain the difference in performance while trying to solve the transfer learning task, namely that the network is not trained well enough to map certain observations. When a trajectory to a certain target has a lot of badly mapped observations, the quality of the state predictions will be worse, which makes it very hard for the reinforcement learning algorithm to solve the task.

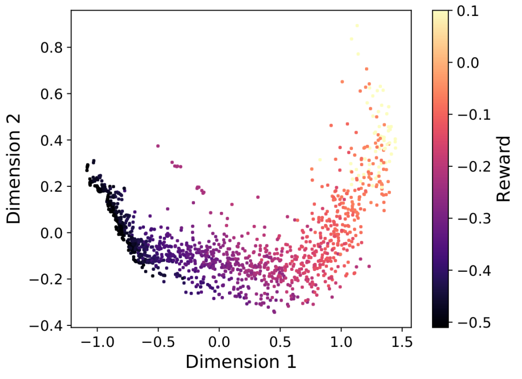

Figure E.1: The state predictions of the *bottom right target* are shaped very good and show a nice gradient in colors. This makes sense since the target is equal to the training situation.

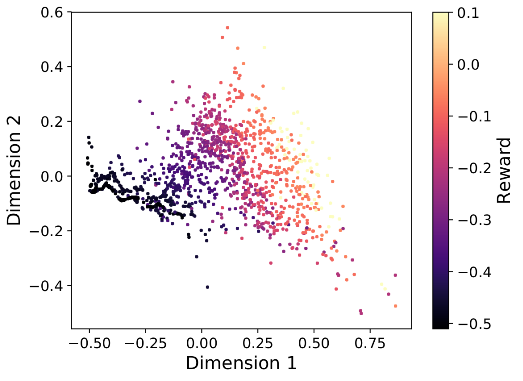

Figure E.2: The state predictions of the *top right target* are shaped less compactly, indicating a little bit more variance in the predictions. The gradient is still good. This explains the good result in general but show why they are not robust (incidental peaks in crash ratio).

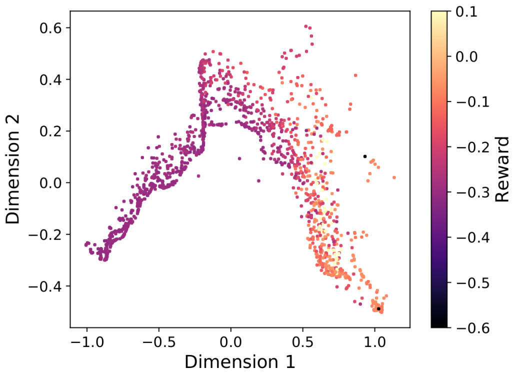

Figure E.3: The state predictions of the *top left target* are very compact and show a very good color gradient, explaining the good results over all.

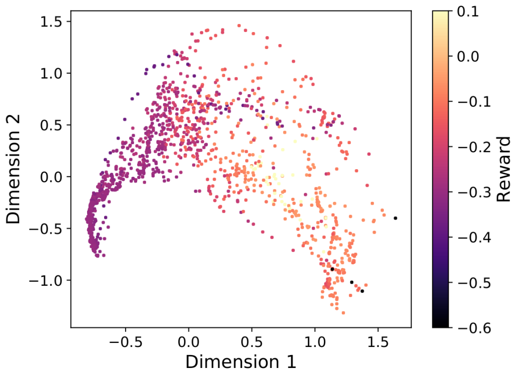

Figure E.4: The state predictions of the *bottom left target* are distorted and are chaotic in terms of colors, resulting in very poor performance.

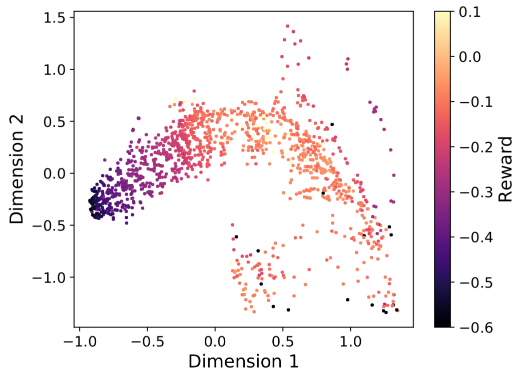

Figure E.5: The state predictions of the *mid right target* show very good properties in the first stage of the trajectories. As soon as the agent approaches the wall the mapping becomes disastrous. This is caused by the fact that the camera captures only a single wall from these positions. Since the network is not trained on these situations, the results are very poor.

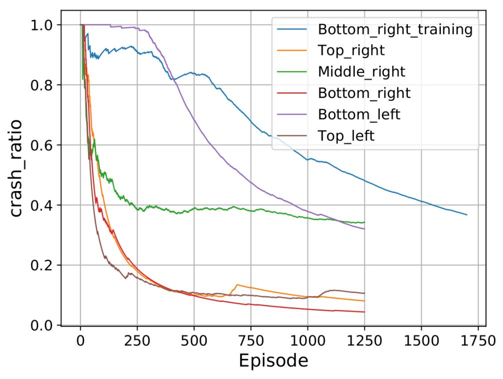

Figure E.6: Crash ratio of multiple reinforcement learning tasks that receive state predictions from a state network trained on the bottom right target (Duplicated to allow for easier comparison).

Bibliography

[1] R. Jonschkowski and O. Brock. Learning state representations with robotic priors. *Auton. Robots*, 39(3):407–428, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[3] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[4] T. Haarnoja, A. Zhou, S. Ha, J. Tan, G. Tucker, and S. Levine. Learning to walk via deep reinforcement learning. *CoRR*, abs/1812.11103, 2018.

[5] S. Gu, E. Holly, T.P. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation. *CoRR*, abs/1610.00633, 2016.

[6] J. Munk, J. Kober, and Robert Babuska. Learning state representation for deep actor-critic control. In *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*, pages 4667–4673, 2016.

[7] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška. Integrating state representation learning into deep reinforcement learning. *IEEE Robotics and Automation Letters*, 3(3):1394–1401, July 2018.

[8] H. Caselles-Dupré, M. Garcia Ortiz, and D. Filliat. Continual state representation learning for reinforcement learning using generative replay. *CoRR*, abs/1810.03880, 2018.

[9] A. Anand, E. Racah, S. Ozair, Y. Bengio, M. Côté, and R.D. Hjelm. Unsupervised state representation learning in atari. *CoRR*, abs/1906.08226, 2019.

[10] T. Lesort, N. Díaz Rodríguez, J. Goudou, and D. Filliat. State representation learning for control: An overview. *CoRR*, abs/1802.04181, 2018.

[11] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[12] H. van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems 23*, pages 2613–2621. 2010.

[13] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[14] L.P. Kaelbling, M. L. Littman, and A.W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[15] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[16] R Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, 11 1954.

[17] K. Arulkumaran, M.P. Deisenroth, M. Brundage, and A.A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34:26–38, 2017.

[18] C. Watkins. Learning from delayed rewards. 01 1989.

[19] Z. Salloum. Double q-learning, the easy way: An intro to understanding double q-learning, 2018. https://towardsdatascience.com/double-q-learning-the-easy-way-a924c4085ec3.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, Rusu A.A., J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[21] A. Choudhary. A hands-on introduction to deep q-learning using openai gym in python. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

[22] Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97 2:285–308, 1990.

[23] L. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, May 1992.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[25] R. Liu and J. Zou. The effects of memory replay in reinforcement learning. *CoRR*, abs/1710.06574, 2017.

[26] D.P. Kingma and M. Welling. Auto-encoding variational bayes, 2013.

[27] C.M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[28] L.J.P. van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-sne. 2008.

[29] N. Botteghi, D.J. Geijs, R. Obbink, M. Poel, C. Brune, B. Sirmacek, A. Mersha, J.B.C. Engelen, and S. Stramigioli. State representation learning with reward-based robotics priors. 2020. Not published yet.

[30] Turtlebot. https://www.turtlebot.com.

[31] Theano deep learning package. http://deeplearning.net/software/theano/.

[32] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[33] Turtlebot 3 e-manual. http://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/.

[34] M. Morik, D. Rastogi, and O. Brock. State representation learning with robotic priors for partially observable environments data. 2019.

[35] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, abs/1703.06907, 2017.