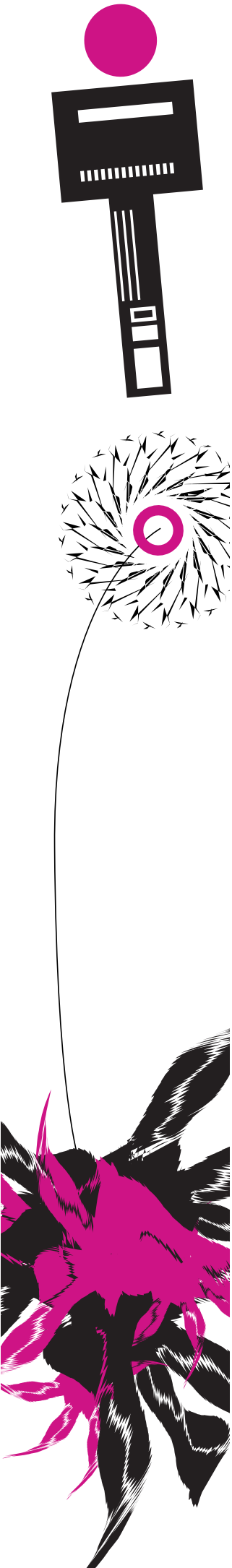


RAM ROBOTICS AND MECHATRONICS

State representation learning using robotic priors in continuous action spaces for mobile robot navigation

A.L. (Arnold) Bijman

MSC ASSIGNMENT

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

August, 2020

040RaM2020
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

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

Abstract

The recent advance of reinforcement learning algorithms has shown that these algorithms are able to solve complex problems. One of these complex problems is the problem of mobile robot navigation. Mobile robot navigation can be highly complex and diverse. To navigate environments, mobile robots must often rely on generic sensors like cameras or lidar sensors. These sensors provide high dimensional data. Using this high dimensional data directly in end-to-end reinforcement learning can be challenging, as this typically requires large amounts of data to learn a task. This is especially problematic in the context of robotics, as acquiring this data can be expensive and time consuming. State representation learning aims to map the high dimensional data from sensors to a low dimensional state space to reduce training time and the required data.

Various methods of learning a low dimensional state representation have been proposed in literature. One method is to use prior knowledge to learn such a representation. This prior knowledge is encoded using loss functions, called robotic priors, which can be used to train an encoder network, implemented using an artificial neural network. This work is focused on mobile robot navigation, where a robot learns to navigate an environment. Previous work from (24) and (11) that has used robotic priors to learn a state representation for the purpose of mobile robot navigation has used discrete actions. This work expands this into a continuous action setting.

The work is done using the Gazebo simulator, in which confined environments are build. A differential drive mobile robot is used for the navigation task. The mobile robot has a camera and a 360-degree lidar sensor to observe its environment. The Gazebo simulator is coupled with python code using ROS middleware. The state representation learning algorithm and the reinforcement learning algorithm are implemented in python using tensorflow. The used reinforcement learning algorithm is DDPG, as this can work with continuous actions and is sample efficient.

To learn a state representation using robotic priors in a continuous action setting, adaptations to these robotic priors needed to be made. This work has shown two ways these robotic priors can be adapted to work with continuous actions. It was shown that these priors can be used to learn a state representation that allows the DDPG algorithm to navigate various environments successfully. Furthermore, the adapted priors are easier to implement compared to the priors introduced in literature and did not require extensive tuning. It is shown that the robotic priors can be used to learn a state representation across a range of simulation environments.

To extend the generality of the state representation learning using robotic priors framework, learning the state representation using robotic priors was extended to learn a recurrent state representation. Environments can be non-markovian, *i.e.* not all observations can be uniquely mapped to the state of the environment. To make the problem markovian, such that it can be solved using reinforcement learning, memory can be added. The work of (36) has introduced recurrent state representation learning using robotic priors. To train the encoder network, this work has relied on ground truth data. In this work it was shown that the previously used priors are sufficient to learn a recurrent state representation.

Acknowledgements

I would like to take this opportunity to thank all the people who have helped me to get to this point.

First of all, I want to thank my daily supervisor, Nicolo Botteghi. Nicolo has been brilliant to collaborate with. He has been very supportive throughout the whole process and was always willing to make time. Whenever I hit a hurdle, he was able to provide clear insight to help me along. His enthusiasm and passion have helped to push me to achieve more. I am thankful for the opportunity he has given me and for his guidance.

I also want to thank the other members of my committee, prof.dr.ir. G.J.M. Krijnen and dr. M. Poel. I thank them for taking the time to read and judge my work.

There are many people who have helped my get to this point, mentioning all of them would be too elaborate. I would like to thank some people in particular. Firstly, I want to thank my parents who have always supported me in any way they could. Their unwavering love and support has meant so much to me. I would not have gotten here without their help and support.

Given the global circumstances, I have been working from home for the majority of this work. I am grateful to my lovely wife for putting up with me. She has helped me get through the challenging periods and has cheered me on tirelessly. Her help and support has helped me stay sane and motivated. Finally, my thanks go to my cat Tobias for keeping me company.

Contents

1 Introduction

1.1 Introduction to the problem context

Mobile robot navigation is an important topic of research. Over the years, many techniques for navigation and localisation have been presented in literature (17). This work will be focused on mobile robot navigation. Environments in which a robot may need to navigate can be highly varied and complex. The problem of mobile robot navigation therefore is not trivial.

One promising approach towards mobile robot navigation is to make use of the recent advances in reinforcement learning. Reinforcement learning in recent years has tackled increasingly complex problems (28), (45). Recent advances have been demonstrated largely in the domain of computer games, with openAIs Dota2 (42) and google deepminds AlphaStar (8) and AlphaZero (47). However, in the field of robotics there have also been promising developments. OpenAI has demonstrated complex dexterity behaviour in a robotic hand that shows similar dexterity to the human hand (41). Although most research has not focused on the problem of mobile robot navigation specifically, these advances could potentially translate to the field of mobile robot navigation.

To make mobile robot navigation more accessible, the robots should preferably make use of generic sensors. This would reduce the cost of building such a mobile robot and can give them a highly diverse functionality. These generic sensors often are cameras, which are cheap and useful for many tasks, as well as lidar sensors to aid in the detection of obstacles. One challenge with using cameras and lidar sensors on mobile robots is that these sensors often produce high dimensional data. This high dimensional data slows down learning significantly, increasing training time and hardware requirements for training a reinforcement learning policy.

One recent approach that has attempted to tackle this problem of high dimensionality in the context of mobile navigation was proposed by (24). Where in the typical deep reinforcement learning setting one neural network is used which maps one high dimensional input to one output, this approach proposes to make use of two separate networks. Instead of mapping the high dimensional input directly to the output, one network is used to map the high dimensional input to a low dimensional state space. This mapping is called the state representation. This low dimensional state space is then used in the reinforcement learning algorithm. Multiple approaches for learning such a low dimensional representation have been proposed in literature (30). The aforementioned approach proposed to use prior knowledge based on physics to train a representation in an unsupervised manner. This approach could reduce training time for reinforcement learning policies significantly, as this method allows high dimensional observations to be mapped to a low dimensional state space. Once such a representation has been trained for an environment, policies can be learned quickly in this environment. These physics based heuristics are implemented by defining a loss function, and are called robotic priors.

This method, as well as other work which has expanded upon this concept (31) has been proposed for a discrete action space, where at each time-step the algorithm can take one of a limited set of predefined actions. In robotics, a continuous action space is often more interesting, as most robotics tasks are continuous in nature. This project aims to test and optimise the learning of a state representation using these robotic priors for a continuous action space. The aim for this work is to allow for quicker learning of new policies and make learning more efficient. In this work, new robotic priors are introduced that enable learning a state representation with a continuous action space.

Furthermore, in many real life problems, the environment in which the robot operates is not fully observable. To be able to learn a policy in a partially observable environment, recurrent neural networks can be used in combination with a reinforcement learning algorithm (19). (36)

has introduced the concept of recurrent state representation learning, where a recurrent neural network is used to learn a state representation, after which a non-recurrent reinforcement learning algorithm can be used for learning the policy. This approach will be further explored in this work.

The devised method should be able to be used as a drop-in replacement for the standard reinforcement learning framework. This means that the proposed method should be based on the information which typically is available in standard reinforcement settings.

1.2 Technical problem overview

The concept of robotic prior based state representation learning was introduced by (24). While different methods of learning a state representation can be found in literature, as summarised in (30), the method of encoding prior knowledge about such an environment to learn the representation space is still under-explored. This method is potentially more efficient than other methods, as this method can achieve a very low dimensional state space representation. Research on this topic (24) (31), (13), (11) has been mostly focused on the discrete action space setting. Given that using continuous actions can give an advantage in robotics as this gives the possibility of more fine grained and precise control, this is seen by the author to be a shortcoming. While there has been work which has used continuous actions, the work of (25) has focused on continuous control tasks instead of navigation tasks. Therefore this work cannot be easily transferred to other tasks, like mobile robot navigation.

The goal of learning a state representation is to significantly reduce the dimensionality of the sensory input data to train reinforcement learning algorithms more efficiently. Cameras and lidar sensors are common and generic sensor modalities that can be used in a wide range of problems. The challenge of working with these sensor modalities is that these often produce highly dimensional sensory data. This high dimensionality is a challenge for learning good policies, as it takes more time to learn a good policy and requires more data when the input is high dimensional. This can be especially problematic in the field of robotics, as obtaining data can be a challenge in real world settings. (24) states that as a robot only need the information relevant to the task, this information can be represented in a low dimensional space. Furthermore, as robots operate in the physical world, and as such are governed by physics, so should the learned state space be constrained by physics. Physics imposes constraints on how the world changes and how the robot can interact with these. Using prior knowledge about interactions with the environment allows to learn representations which are consistent with physics. These priors further allow to learn the relevant information, while discarding information that is not relevant for the task for which the representation is learned. The way this prior knowledge is implemented is through loss functions, the loss is then minimised by optimising the state representation network.

This work considers the case of mobile robot navigation. The mobile robot makes use of differential drive for manoeuvring. The robot has a generic RGB camera at the front, which gives a 62.2 degrees field of view. In addition to the camera, the robot also has a lidar sensor, which has a 360 degrees field of view, and a maximum range of 3.5 meters.

Using only information which is typically available in the reinforcement learning setting, the state representation will be learned. This work will focus on learning a state representation when making use of a continuous action space. This makes learning such a representation more challenging. To learn the state representation, experiences can be gathered from the environment.

This work will be done in a simulation environment. The gazebo simulator is used, the mobile robot which is used in the simulation environment is the Turtlebot3 waffle pi mobile robot (4).

1.3 Project objectives and topics

This work will be focused on two main parts. The first part will be to evaluate the effectiveness of the priors from literature in a continuous actions space and to optimise the learned representation using these priors in the context of continuous actions. This part will adapt priors from literature to improve the learned state representation when using continuous actions. The second part of this work will be focused on learning a state representation in the partially observable setting. Many real life problems are partially observable, or non-markovian, and thus learning a state representation in a partially observable setting is relevant for applying this method to real world problems. While one work which learned a state representation (36) has been investigating the partially observable setting when learning the state representation using robotic priors, their work made use of ground truth data to learn this representation. This work will focus on finding a good state representation without using any data that would not be available in the standard RL setting. Based on these objectives we can formulate two research questions:

1 How can robotic priors be used to learn a state representation in a continuous action space?

The set of robotic priors which were found to be most effective for learning a good state representation will be used for learning the state representation in the partially observable setting.

2 To what extend can a fully observable state representation be learned using the adapted priors to train a recurrent state representation encoder?

1.4 Report outline

This report will first expand on some of the theory which will be used in this work. Background information will be given on the reinforcement learning method which is used. The different methods for learning state representations will be further expanded upon and explained. Then an analysis will be given on the different parts of the overall setup of the experiments. First the state representation learning using robotic priors will be further introduced, and several different priors will be discussed. Then different elements of the reinforcement learning algorithm will be given and the connection between the different parts of the system will be explained. After the analysis, the design and implementation of the system and the performed experiments will be discussed. The results of the experiments will then be presented and analysed. Finally, a conclusion is given and recommendations for further work are made.

2 Background

2.1 Reinforcement learning

Reinforcement learning deals with the problem of learning from trial-and-error. Unlike dynamic programming, RL does not assume full knowledge of the environment beforehand. Instead, for an RL agent to be able to learn it has to be able to interact with the environment, and collect information from the environment. Where supervised learning problems learn from some ground truth data which is provided to the algorithm, in the reinforcement learning setting no such ground truth is available. Instead, learning is done using feedback signals from the environment.

Generally speaking, the RL problem is a discrete time stochastic control process. The agent interacts with its environment in discrete steps. The agent starts in a given state within the environment $s_0 \in S$. An initial observation $\omega_0 \in \Omega$ is gathered. At each time step t, the agent takes an action $a_t \in A$. This action gives three consequences: the agent obtains reward $r_t \in R$ from the environment, the next state s_{t+1} and the next observation ω_{t+1} resulting from this state. The agent can use these elements to learn a policy π. An overview of the RL setting can be seen in Figure 2.1.

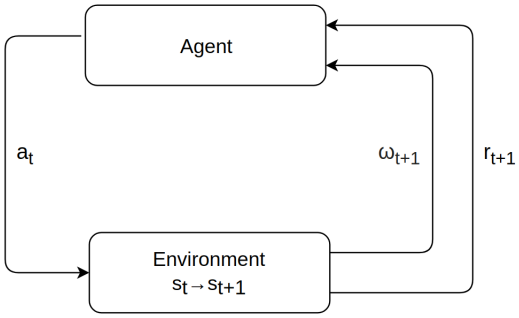


Figure 2.1: Interaction between agent and environment in the RL setting

2.1.1 Markovian environment

In the typical RL setting, the environment is considered to be Markovian, or fully observable. The problem is Markovian if

- $\mathbb{P}(\omega_{t+1}|\omega_t, a_t) = \mathbb{P}(\omega_{t+1}|\omega_t, a_t, ..., \omega_0, a_0)$
- $\mathbb{P}(r_t|\omega_t, a_t) = \mathbb{P}(r_t|\omega_t, a_t, ..., \omega_0, a_0)$

The Markov property means that the next observation and reward only depend on the current observation and the action the agent takes after this observation. If the Markov property holds, the RL problem becomes a Markov Decision Process (MDP) (10). This means that the agent is not interested in the history of observations and actions, as the current observation fully summarises the state of the environment. This means that if the system is fully observable $\omega_t = s_t$. The MDP is then defined by the tuple (S, A, T, R, γ), where S is the state space, A is the action space, T is the state transition function between states, R is the reward function and γ is the discount rate, with $\gamma \in [0, 1]$. At each item step t, the probability of moving to s_{t+1} is given by the state transition function $T(s_t, a_t, s_{t+1})$ and the reward is given by $R(s_t, a_t, s_{t+1})$. The state space S and the action space A can both be continuous or discrete.

Tasks in the real world often are not fully observable, but instead are partially observable. In the partially observable setting, the problem changes to a Partially Observable Markov Decision Process (POMDP) (20). In this setting, the observation $\omega_t \neq s_t$. The POMDP is defined by the tuple $(S, A, T, R, \gamma, \Omega, O)$. Here, S is the state space, A the action space, T the state transition function, R the reward function and γ the discount rate as in the MDP setting. in the POMDP setting the observation is not longer equal to the state, and thus the agent receives an observation $\omega \in \Omega$. This observation is generated from the underlying system state s according to probability distribution $\omega \sim O(s)$. An environment is partially observable when an observation is not sufficient to uniquely identify the underlying state. In the presented theory, the Markovian property will be assumed. RL algorithms have no explicit means of dealing with partially observability. This can hurt performance. To better estimate the underlying system state, memory can be added in the form or a recurrent neural network to include past observations. Another approach to deal with a partially observable environment is to use multiple observations.

2.1.2 Common concepts

There are some aspects of RL which are common across all methods. Here the terminology will be explained.

Environment, agent, and goal

The RL problem deals with an agent with policy π, which is placed in an environment ξ. The agent is given a goal g_ξ. The goal can both be static or dynamic. In case the goal is dynamic the goal typically is part of the state. These goals can be very diverse, ranging from moving to a target, to picking up an item, to solving a level of a game. The agent learns to reach the goal, or solve the problem by making use of the reward function $R(s_t, a_t, s_{t+1})$. The environment or task the agent has to solve can both be continuous and episodic. Continuous tasks do not stop when a goal is reached, instead these tasks can go on forever. An example of a continuous task is predicting stock action. Episodic tasks are more common. An example of episodic tasks are the classic Atari games, where the task ends when the game level is finished or the agent dies.

Policy

The policy π defines the behaviour of the agent given the environment and the goal. The policy maps the observations from the environment to actions. These policies can come from a lookup table or a simple function, where in other cases the policy can be more involved. These policies may be either deterministic or stochastic. The policy will generally try to maximize the cumulative reward the agent receives.

Reward function

The reward signal r allows the agent to learn a task. The reward signal thus specifies the goal of the agent implicitly. Each time step, each step an agent has performed an action, the agent receives a reward. This reward is a simple scalar value. The policy will be optimised to maximize the total reward the agent will receive in the long run.

The reward may either be dense or sparse. In the dense reward setting, the agent receives a non zero reward every time step. If the agent has to move to a target, this may be a reward based on the distance of the agent to the target. In the sparse reward setting, the agent receives zero reward most of the time, and -1 if it bumped into some wall or did a counterproductive action, and +1 when it has reached the goal. A dense reward setting allows for quicker learning, as the agent has a more straight forward way of getting more reward. In the sparse reward setting, the agent has to first get a positive reward by accident, after which it can start learning. A dense reward therefore often offers better and quicker convergence. However, when the task is

simple, like navigating to a goal, constructing this dense reward function can be quite straight forward, in more complex environments or for more complex tasks, this is not so easy.

Value function

The reward indicates how good a state is in the immediate sense. The value function indicates how good a state is in the long run (49). The value of the state is not just the reward it receives at that state, but it is a sum of all discounted future rewards it expected to receive when in that state, following policy π.

The goal of the agent is not to receive the maximum reward at each step, but instead to maximize the total reward. The value function evaluates the value of each state, which is roughly the amount of reward the agent is expected to receive in the future starting from that state. Reward therefore is the short term value of a state, where the value is rather the longer term value of each state. Some states can then get low reward, but high value because the goal is very likely to be reached from this state. The value function can be implemented using a table, but for larger state spaces this becomes intractable. Therefore, the value function is typically implemented using a neural network.

Using the intuition that reward now is often more valuable than the reward somewhere in the future, the reward often is discounted by the discount rate γ. Equation 2.1 gives the value function. Here r is the reward, and K the terminal state.

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{K} \gamma^{k} r_{t+k} | s_t = s, \pi\right] \tag{2.1}$$

As estimating the value function from the rewards is not a straight forward task, the way in which this value function is estimated is an essential part of many algorithms. The value function represents the value of a state, or the discounted sum of rewards the agent can expect. Since the agent tries to find the policy π to maximize the sum of rewards, the agent must seek the state with the highest value. The value function can thus be used to learn a task. The optimal expected return is given by Equation 2.2

$$V^{*}(s) = \max_{\pi \in \Pi} V^{\pi}(s) \tag{2.2}$$

On policy vs off policy

Reinforcement learning can both be on and off policy. Off policy RL algorithms are algorithms which can learn their policy both with samples obtained from the current policy, as well as samples from a different policy. This means in practice, that off policy algorithms can reuse samples which are collected in an earlier state of training, typically by a different policy. This reusing of older samples makes off policy algorithms more sample efficient, they require less samples than on policy algorithms to converge. On policy algorithms on the other hand, can only train their policy using samples collected by their current policy. Off policy algorithms make use of experience replay to learn from past experiences.

Experience replay

A sequence of experiences often has high correlation between the samples. This high correlation between the samples would negate the benefit of using batch updates. Furthermore, the neural networks are prone to catastrophic forgetting (27), a property of neural networks to forget previously learned behaviours when optimised to learn new behaviours. When the network is updated using sequential data from the agent, the network could forget values that occurred early in the sequence when optimised only for the values late in the sequence. To make the updates stable and prevent catastrophic forgetting, DQN introduced experience replay.

When interacting with the environment, the agent collects experiences. Each experience can be defined by the collected tuple $(s_t, a_t, r_{t+1}, s_{t+1})$. These experiences can be added to a replay memory R with capacity N. This memory should span multiple episodes if the task is episodic. When a new experience is recorded it can be added to memory. When the memory is filled up to N, an earlier experience can be removed and updated with the new experience tuple. Each update, a minibatch of experiences can be drawn $e \sim R$, which can then be used to update the network parameters θ. This allows for each experience to be used multiple times for updating the network, which increases sample efficiency.

2.1.3 Most important RL methods

Value based methods

In addition to the value function, the action-value function can also be used. This function, instead of representing the value of each state, it represents the value of each state-action pair. Equation 2.3 gives the Q-value function.

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{k=0}^{K} \gamma^k r_{t+k} | s_t = s, a_t = a, \pi\right] \tag{2.3}$$

This Q-value function can also be written recursively using the Bellman equation:

$$Q^{\pi}(s,a) = \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma Q^{\pi}(s', a = \pi(s')) \tag{2.4}$$

Similarly to the value function, the optimal Q-value function can be defined as:

$$Q^*(s,a) = \max_{\pi \in \Pi} Q^{\pi}(s,a) \tag{2.5}$$

The optimal policy can be directly obtained from $Q^*(s,a)$:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s,a) \tag{2.6}$$

In the basic version of Q-learning, this Q-value function is implemented using a lookup table, with an entry for each state-action pair. As shown in equation 2.6, the optimal policy can be obtained by taking the action with the maximum Q-value. Equation 2.4 shows the equation for getting the Q values, however this equation depends on transition function $T(s,a,s')$, which is not available to the agent, as the agent does not have an underlying model of the environment. Instead, the Q-values can be estimated recursively, using equation 2.7.

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_{t-1}(s_{t+1}, a) - Q_{t-1}(s_t, a_t)) \tag{2.7}$$

Here α is the learning rate. The updating rule makes use of the Bellman equation. The intuition behind this equation is the following. Given equation 2.3, the following holds:

$$Q^*(s_t, a_t) = \mathbb{E}\left[r_{t+1} + \gamma Q^*(s_{t+1}, a_{t+1})\right] \tag{2.8}$$

This should hold for the Q-function, as the total discounted reward the agent receives from state s_t should be equal to the reward in state t plus the total discounted reward from state s_{t+1}. If this is not the case, the Q-function is not fully converged to its optimum value. As Q-learning follows the simple greedy policy $\max_a$, this methods can be applied off-policy. Tabular Q-learning is shown to converge, given enough samples for each state, and a discrete state-action space (51).

When the state-action space is large, as it often is in more complex environments or with more complex observations, using the tabular method becomes impractical. The big state-action

space will prevent effective training, as each state needs to be visited multiple times to reach the optimal Q-value function. To deal with these huge state-action spaces, function approximators can be used to approximate the Q-function instead of keeping all values in a lookup table. A commonly used function approximator is an artificial neural network, which has shown state of the art results. A popular algorithm using a neural network as function approximator is the DQN algorithm (34).

Instead of estimating $Q(s,a)$ directly, DQN makes use of a neural network with parameters θ, now θ needs to be found such that $Q(s,a;\theta)$ converges to the optimal Q value. . Similarly to tabular based Q-learning, the Bellman equation of 2.8 may be used for learning the $Q(s,a;\theta)$. A loss function may be defined:

$$L_i(\theta_i) = \mathbb{E}_{,}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{2.9}$$

with $y_i = \mathbb{E}_{'}\left[r + \gamma \max_{a'} Q(s',a';\theta^-)|s,a\right]$ is the target value for iteration i . The target depends on the network parameters θ and therefore change during training. This makes training prone to instability. To make training more stable θ^- is used, which is an older version of the network, which, unlike the main network θ is updated only every k iterations, to $\theta^- = \theta_i$. The network parameters are updated using gradient descent, using the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{,}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{2.10}$$

In practice it is impractical to compute the full expectation. Instead, a batch of experiences can be used. Although a single experience could be used for updating the network, using a batch update, in which the individual experiences are averaged, reduces variance and allows for larger updates.

In the basic Q-learning implementation, a lookup table is used with an entry for every state-action pair. Using the state and action as input to the network would be impractical, as one would have to compute the output of the network for every action for each state. Instead, the state is used as input to the network, and the network has a separate output for each of the actions. This can be seen demonstrated in Figure 2.3.

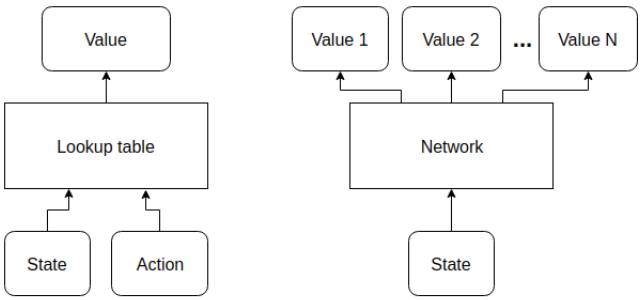


Figure 2.2: The basic architecture for Q-learning based on a lookup table (left) and DQN (right). The DQN has one value output for each action, giving the value of that state-action pair

Policy gradient methods

This family of RL algorithms optimize a performance objective by finding a good policy. There exists a group of RL algorithms that optimize the policy without using gradients, like evolutionary methods (43), but using gradients is more common. Policy gradient methods can be both deterministic and stochastic. Stochastic policy gradient methods do not output actions directly, but instead output a probability distribution over the actions, from which one of the

actions is selected according to this distribution. Deterministic policy gradient methods instead are adapted to be able to deal with continuous actions. Some deterministic methods are DDPG (48) and NFQCA (18). Policy gradient methods can be optimised without estimating a value function, by means of Monte Carlo estimates. This means that instead of estimating the value function, the empirical reward from a trajectory is used as value estimate. This method is used in the REINFORCE algorithm (53). This Monte-Carlo estimate is an unbiased estimate, as it uses the true sum of rewards the agent has received. It is however high variance as the observed rewards can differ significantly between episodes. This method also requires a full episode roll-out to collect the empirical reward, which makes this method slow to converge.

A more efficient way to estimate the value of states is to use the actor-critic architecture (29). This architecture has two parts, an actor and a critic. The actor represents the policy, while the critic represents the estimation of the value function. Typically, the actor and critic are represented using neural networks. The value function can be estimated similarly to value function based methods. The actor, or policy can be updated similarly to pure policy based methods, using the critic's value function estimate instead of the Monte-Carlo style value estimation. The actor-critic method can be used both in the on-policy and off-policy setting. In the off policy method, the gradients are generally biased, as the policy used to collect the samples is generally different from the policy which is updated. Using data stored in memory makes these off-policy methods more data efficient, but less stable as they have biased gradient updates. On policy methods are less sample efficient, but more stable due to the unbiased gradient updates. Typically, on policy actor-critic algorithms uses parallelization of agents to ensure the agent experiences different parts of the environment at each given time step.

DDPG is a policy gradient method designed specifically to work with a continuous action space. Other policy gradient methods can also be used in a continuous action space, these include PPO (44), A2C and A3C (33). These methods are actor-critic methods, like DDPG. The critic in the actor-critic methods learns the Q-value function of the environment and serves as a critic for training the actor network. PPO, A2C and A3C are on policy methods, meaning that the actor and critic are trained only using data generated by the current policy. DDPG is an off policy method, and makes use of a replay memory. This makes the DDPG algorithm more sample-efficient, as the replay memory allows samples to be used for multiple updates. The DDPG algorithm has been tuned for better performance (15). This however introduces more complexity in the algorithm. The DDPG algorithm is considered to be relatively simple to implement, and can in many cases reach a good performance.

Deep Deterministic Policy Gradients (DDPG)

The DDPG algorithm, as proposed by (32) is a deterministic policy gradient method. The DDPG algorithm uses an actor-critic architecture, with one neural network representing the policy μ and the other neural network representing the value function θ. The policy network outputs the actions directly instead of a distribution over the possible actions. This makes the DDPG algorithm work with continuous action spaces. Similar to Q-learning, the DDPG algorithm uses the Bellman equation to update the action-value function estimate. DDPG is an off-policy algorithm, meaning that its policy can be trained using samples generated by a different policy. DDPG, like DQN makes use of experience replay to decorrelate the training samples and stabilise learning. Experiences are saved in a replay memory R, as experience tuples $(s_t, a_t, r_{t+1}, s_{t+1})$.

In Q-learning, the value function approximation can be updated using the loss function in equation 2.9. This equation makes use of the greedy policy $\pi^*(s) = \arg\max_{a \in A} Q^*(s,a)$. As the actions are continuous instead of discrete like in Q-learning, using such a greedy policy would require an optimization over the entire action space. Instead of using such an optimization, a policy network is used to obtain the action.

Q-learning makes use of a target network $Q(s,a;\theta^{-})$, in the target estimation in the bellman equation to make learning more stable. Instead of copying the Q network every k time steps, the DDPG algorithm uses soft target network updates. This means that a copy of the actor and critic networks is created, $\pi(s;\theta'_{\mu})$ and $Q(s,a;\theta'_{Q})$, and updated each training step. The values of these target networks are then updated using polyak averaging: $\theta_{target} = \rho\theta_{target} + (1-\rho)\theta$. ρ is a hyperparameter between 0 and 1. The loss for the value function estimate, from which the gradient update can be computed, is be defined as:

$$L_i(\theta) = \mathbb{E}, \left[\left(r + \gamma Q(s', \pi(s';\theta'_{\pi});\theta'_{Q}) - Q(s,a;\theta_{Q}) \right)^2 \right] \tag{2.11}$$

The deterministic policy $\mu(s;\theta_{\mu})$ is optimised to give the action which maximizes $Q(s,\mu(s;\theta_{\mu});\theta_{Q})$. Since the action space is continuous maximizing Q is problematic since it requires a global maximisation at every step. Instead, the policy can be moved in the direction of the gradient of Q, instead of globally maximizing Q. Since the Q function is differentiable, the policy parameters can be updated using gradient ascent. The parameters θ_{π} can be updated using gradient $\nabla_{\theta_{\pi}} Q(s,\pi(s;\theta_{\pi});\theta_{Q})$. Using the chain rule, this update becomes:

$$\mathbb{E}\left[\nabla_{\theta_{\pi}} \pi(s;\theta_{\pi}) \nabla_{a} Q(s,\pi(s;a);\theta_{Q})|_{a=\pi(s;\theta_{\pi})} \right] \tag{2.12}$$

A challenge when using a deterministic policy is exploration. When using a stochastic policy, the policy will explore due to the stochastic nature of the policy. With a deterministic policy, noise can be added to the output to aid exploration. The original work used the Ornstein-Uhlenbeck process (50) to generate temporally correlated noise.

$$\pi'(s_t) = \pi(s_t;\theta_{\mu}) + \mathcal{N} \tag{2.13}$$

Algorithm 1: Overview DDPG

Initialise actor and critic networks $\pi(s;\theta_{\pi})$ and $Q(s,a;\theta_{Q})$ with random weights
Initialise target networks as: $\pi(s;\theta'_{\pi}) \leftarrow \pi(s;\theta_{\pi})$, $Q(s,a;\theta'_{Q}) \leftarrow Q(s,a;\theta_{Q})$
Initialise replay memory R
for *episode in N* **do**
  Receive initial observation o_0
  Compute $\hat{s}_0 = \phi(o_0)$
  Initialise OU noise $\mathcal{N}$
  for *t in max_steps* **do**
    Compute action $a_t = \pi(\hat{s}_t|\theta_{\pi}) + \mathcal{N}$
    Take action a_t and observe r_{t+1}, o_{t+1}
    Compute $\hat{s}_{t+1} = \phi(o_{t+1})$
    Save experience tuple $(s_t, a_t, r_{t+1}, s_{t+1})$ to R
    Sample a minibatch with N experience tuples from R
    Compute $y_i = r_{i+1} + \gamma Q(\hat{s}_{i+1}, \pi(\hat{s}_{i+1};\theta'_{\pi});\theta'_{Q})$
    Update critic network using loss$L = 1/N \sum_i \left(y_i - Q(\hat{s}_i, a_i;\theta'_{Q}) \right)^2$
    Update actor network according to
     $1/N \sum_i \left(\nabla_{\theta_{\pi}} \pi(s_i;\theta_{\pi}) \nabla_{a} Q(s_i, \pi(s_i;a);\theta_{Q})|_{a=\pi(s_i;\theta_{\pi})} \right)$
    Update the weights of the target networks:
    $\pi(s;\theta'_{\pi}) \leftarrow \rho\pi(s;\theta_{\pi}) + (1-\rho)\pi(s;\theta'_{\pi})$
    $Q(s,a;\theta'_{Q}) \leftarrow \rho Q(s,a;\theta_{Q}) + (1-\rho)Q(s,a;\theta'_{Q})$
  end
end

2.2 State representation learning

As discussed in the previous section, current RL methods have shown to be able to learn impressive behaviours from raw pixel inputs. Such high dimensional input however comes at the cost of increased computational load and reduces the sample efficiency, taking more time and samples to converge. While Atari games can be played with an input image size of 64x64 pixels (35), other tasks might require more detailed inputs. The number of input dimensions then quickly increases, and with it the training time. With an increased input dimensional, the model typically needs to be more complex to handle the larger inputs. This gives an increased number of parameters to match to an increased number of inputs.

Often, the information present in the raw observation can be represented using a limited number of variables, which contains all information relevant for solving the task. Take a navigation task in a known environment. When using a camera for observing the environment, with o parameters per observation, the agent can learn to navigate this environment using the camera. But the only information the agent needs to successfully navigate the environment would be its position and orientation. While more complex tasks would require an increased number of inputs, the minimum number of dimensions d is usually $d << o$.

Traditionally, in robotics applications, the observation space is designed manually. While this is often possible for easier tasks, when more complex tasks are considered, designing the observation space is non-trivial. Furthermore, when in simulation one often has access to the true state. When executing a task in the real world, one must rely on sensory data. Most applications must rely on generic sensors, like cameras or lidar sensors. These sensors produce high dimensional data, which makes them very versatile, but also more difficult to work with. Extracting the relevant information from this high dimensional sensory data is not a trivial endeavour.

Reinforcement learning algorithms have shown to work well when having only raw pixels as input. The neural networks then develop an internal state representation, which represents the relevant information in lower dimensionality. Learning such a representation is done explicitly by optimising for the task learned. Such a learned representation is not interpretable as this is often formed within a neural network. When a new task is taught to the reinforcement learning algorithm, even if this taks is within the same environment, the learned internal state representation cannot be re-used but must be learned again for each task. For mobile robot navigation specifically, this means that learning to reach a new position in the environment means learning a new representation of the environment.

To make a learned representation more interpretable as well as reusable, state representation learning (SRL) can be used. SRL aims at learning the underlying state from observations, in an unsupervised or self-supervised manner. This observation state mapping is learned in the early phase of training, and then the state is used as input to the RL algorithm. SRL aims at learning this mapping such that the problem is a MDP, such that the states are not ambiguous. To learn the observation-state mapping ϕ, all observations up to that state, the actions taken and the reward received from the environment can be used, as described in equation 2.14.

$$\hat{s}_t = \phi(o_{1:t}, a_{1:t}, r_{1:t}) \tag{2.14}$$

While traditional RL learns the relevant state implicitly, from observations using only the reward signal, SRL aims at learning the state representation more efficiently by making use of additional information and constraints.

Learning this state is a non-trivial task, as the state should contain all information which is relevant for the task. At the same time, the number of variables in the state must be kept at a minimum as to keep the training of the agent efficient, so the representation should be able to represent the state with a minimal number of variables. If the environment is fully observable,

e.g. markovian, then so should the state representation be, creating a unique state representation for each of the observations. Apart from being markovian, the representation should represent the true value of the state good enough to be able to make policy improvements based on the representation. Furthermore, as it would require too much training time to collect all possible states, the observation-state mapping should generalise to unseen states which are similar to the previously seen states.

Several methods for learning a state representation have been developed over the years. Three major approaches can be distinguished. These approaches will be discussed briefly.

The first approach is to use compression of observations. This approach uses methods for compressing the observations to a lower dimensional observation space. This can be achieved using a linear method like PCA (14), but is typically achieved using the auto-encoder framework (21; 52; 22). The main downside of this approach is that the auto-encoder framework does not distinguish between task relevant information and task irrelevant information. Therefore, part of the learned state space is used to encode information which is not relevant for solving the task. Furthermore, this framework is known for ignoring smaller objects present in the observations, while these objects can be relevant for the task.

The second approach is to learn a dynamics model, which learns to model the dynamics of the environment (7). One way to achieve this is to use the forward model, in which the next state is estimated using the current state. The difference between the predicted next state and the encoded next state is then minimised. The inverse model instead tries to predict the action a_t which was taken to get from $\hat{s}_t$ to $\hat{s}_{t+1}$. The forward and inverse models have been found to be complementary and can be used in parallel to learn a good representation. In many works these dynamics models are combined with the auto-encoder framework in order to improve and stabilise the learned state representation. More information on both the compression of information and dynamics model methods can be found in A.

Finally there is the method of using robotic priors for learning the state representation. This method was devised specifically for the context of robotics, in which the changes in the environment are confined by the laws of physics. This framework uses loss functions, which encode some prior knowledge of the environment to learn the state representation. Unlike the compression of observations and the dynamics model approaches, using robotic priors allows the state representation to be learned using the reward signal from the environment. This allows this method to learn a representation which encodes information that is relevant for the task, and ignore information which is not relevant for the task. This is achieved by the fact that information which is relevant for solving the task is reflected in the reward signal. The robotic prior approach assumes that all task relevant information can be contained in a few dimensions. Reducing high dimensional observations to only a few dimensions allows for learning the task more efficiently using reinforcement learning.

2.2.1 Learning a state representation using robotic priors

(24) introduced the concept of robotic priors for learning state representations. Unlike priors in the context of Bayesian learning, priors do not refer to a probability distribution, but rather a loss function representing some knowledge about the world or task. The goal of these priors is to learn low dimensional representations from high dimensional input data in an efficient manner. In their work, it was shown that using priors it was possible to learn a state representation which corresponds to physical properties. These priors exploit the fact that the robot acts in the physical (simulated) world, and therefore it has to adhere to the laws of physics. While the previously discussed methods learn state representations based on predictions about either future or past values, or try to compress the observations, robotic priors make use of the distance between two states to learn a consistent and coherent state representation.

The priors introduced in the work of (24) all assume the Markovian property, such that the state can be deduced from a single observation. The priors are thus used to learn the the mapping from observations to states $s_t = \phi(o_t)$. Learning the state representation is formulated as an optimization problem, where the priors are implemented as loss functions. The network ϕ is then trained using the combined loss from each prior. To compute the loss, data from the agent interacting with the environment is used $D = \{o_t, a_t, r_t\}_{t=1}^{n}$, which consists of observations from the sensors of the robot, the action the robot took, and its reward. As learning the state representation is formulated as an optimization problem, the mapping is robust to counter examples for each prior, it simply learns the mapping which is most consistent with each prior. To compute the loss, a Siamese network architecture is used, where two observations are passed through two identical networks. These network share all parameters. The resulting states for both observations will then be used for computing the loss on the prior. Data from the environment is collected either using an RL policy or by random exploration, and added to a memory which saves the experiences.

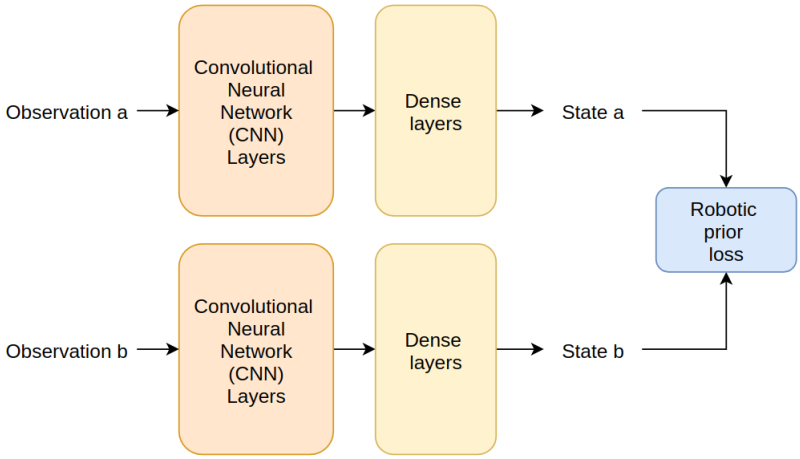


Figure 2.3: The siamese network architecture which is used to compute the loss on the priors. All parameters are shared between the two models

Neural networks are typically trained using minibatches to reduce variance and therefore increase stability. To compute the loss for each of the priors, a minibatch of collected experiences is sampled from memory. For each of the priors which is used, the loss on this minibatch is computed. The network is then trained using gradient descent to reduce this loss.

2.2.2 State of the art for learning a state representation using robotic priors

The topic of learning a suitable state representation using robotic priors has not been researched extensively. The work of (24) has introduced the topic. In their work ϕ, with $\hat{s}_t = \phi(o_t)$ was constrained to be linear.

The first task on which their method was applied, was a simple navigation task. In this task, a mobile robot had to navigate a small square environment to reach a target location. In this small square environment, each of the walls had a different color. The mobile robot could navigate forward, backward, left and right at each time step. Several different, discrete step sizes could be chosen. The orientation of the robot was kept fixed, limiting the observation space of the robot. The robot has a camera to obtain the observations, this camera has a field of view of 300^o. Using a top-down camera view instead of a camera on the mobile robot also worked well.

The second environment on which their method was applied was a slot racing car challenge. This challenge had two cars, of which only one was controlled. The observation showed the

entire environment, including the two cars. The learned state representation was able to ignore the car which could not be controlled.

In a third experiment the simple square environment was revisited. This time, moving circles and squares of a different color were added to the floor and the walls respectively. It was found that these distracting shapes did not have a negative impact on the learned state representation.

In later work (25), the position-velocity encoder was introduced. This work focused on continuous control tasks, in which the environment dynamics are most important. This work proposed to compute a velocity state explicitly using finite differences between the position states. This work introduced priors which could make use of this explicitly computed velocity state. using this position-velocity encoder, the inverted pendulum, cart pole and ball in cup tasks could be learned from pixel inputs. This work used a continuous action space. The three tasks on which this method was evaluated are similar in structure, and it is not clear how this method would transfer to other tasks.

(31) has expanded the use of robotic priors for learning a state representation into the domain of robotic arms. In this work, a neural network is used to learn the state representation from observations. These observations are images from a fixed camera. For this work the baxter robot (1) is used. The agent is trained to push a button using the end-effector of one of the arms of the robot. The other robot arm is used as a distractor object. It was found that this robot arm could be ignored if it was moving, but when the distractor robot arm was kept still within an episode, but at different positions each episode, the learned state representation suffered. By introducing an additional prior, this could be mitigated.

The work of (36) has focused on learning a state representation in a partially observable setting. For this work, a 3D maze environment was created. This environment was partially observable. In order to learn a good state representation, a recurrent neural network was used to map the ambiguous observations to unique states. To be able to learn a good representation, an additional prior was introduced.

2.2.3 Summary

This chapter has covered the relevant background information which is needed to read this work. First, the reinforcement learning framework was introduced. This covered some of the main concepts that come with the reinforcement learning framework. Concepts like the value function, the policy and the reward function and experience replay were discussed in detail. Then the basics on the value based methods and the policy gradient methods were introduced and explained. The DDPG algorithm was then considered in more detail.

After introducing the reinforcement learning framework, the state representation learning framework was introduced. This has covered the aim and motivation for learning such a state representation. One method of learning a state representation is to use robotic priors. This concept was introduced in more detail. Learning a state representation using robotic priors entails training an encoder network using the robotic priors, which are defined by loss functions. These loss functions encode some prior knowledge on how state space should be shaped, based on laws of physics. A Siamese architecture is used for computing these losses.

Finally, the current state of the art for learning a state representation using robotic priors is explained. This has covered the various works from literature that has used the concept of robotic priors for learning a state representation.

3 Analysis

This work aims to improve upon the state of the art of learning a state representation using robotic priors. The application to which this will be applied is mobile robot navigation. This work will focus mainly on two separate areas. The first part of this work is focused on learning a state representation using robotic priors in a continuous action space. The second part of this work will expand learning a state representation to more difficult problems using a recurrent state representation encoder network.

3.1 Research aims

Previous work on state representation learning using robotic priors has mostly been focused on environments with a discrete action space. (36) has applied robotic priors for learning a state representation in a continuous action space, however, this work used ground truth information to learn the state representation. It is our goal to use only the information that is generally present in the reinforcement learning setting. Another work which has used robotic priors with a continuous action space is (25), however this work did not focus on the problem of mobile robot navigation. It is our belief that further research into learning a state representation using robotic priors in a continuous action setting can make applying reinforcement learning more practical. Using a continuous action space allows for more fine-grained control. In the context of robotics this is especially relevant, as many robotics problems are continuous in nature.

While the previously mentioned works have used robotic priors to learn a state representation in a continuous action space, these works cannot be directly applied to the problem of mobile robot navigation. The work by (24), which introduced the concept of robotic priors has used these priors to perform mobile robot navigation. (11) has adapted these priors and expanded the problem into more challenging environments. This inspires the first research question:

1 How can robotic priors be used to learn a state representation with a continuous action space?

This raises the question if the priors which have been proposed in literature are able to learn a good representation if these are adapted to a continuous actions space. Therefore the question:

1a Are the proposed priors from (24) and (11) able to learn a good representation when using a continuous action space, when these are adapted to better work with continuous actions?

The priors proposed in (24) and (11) are conditional, meaning that pairs of training points need to be found that meet these conditions. As the action and reward space are both continuous, it might be possible to construct priors which are fully continuous. This would remove the need to find matching pairs of points for training. Finding these points can form a performance bottleneck, and removing these conditions not only improve performance but will also make the priors easier to implement. Getting rid of the conditions in the robotic priors can reduce the number of hyper-parameters, which might make the robotic priors more robust to different environments or easier to tune. Therefore the question:

1b Can the current sets of conditional priors be adapted to better make use of the continuous nature of both the reward signal and action space by removing conditioning for training pairs?

After evaluating different priors, these priors can be used to move into the more complex partially observable setting. The work of (36) has considered state representation learning in a

partially observable setting. In their work they have not touched upon the originally proposed priors, neither on adapted variants of these. It remains to be seen if these priors, or adaptations of these can be made to work in a partially observable setting. Therefore the question:

2 To what extend can a recurrent state representation be learned using the adapted priors in a continuous action space?

This work will investigate the presented research questions, and attempt to answer these questions.

3.2 System analysis

This section will cover the main components of the overall system. First, learning the state representation learning using robotic priors will be analysed. After this, a further analysis of the used reinforcement learning method is given. Lastly, an analysis of the used mobile robot is given.

3.2.1 State representation learning using robotic priors

As mentioned previously, learning the state representation means finding the mapping $\hat{s}_t = \phi(o_t)$ from high dimensional o_t to a low dimensional $\hat{s}_t$. This low dimensional state $\hat{s}_t$ should contain information about the state of the environment, and needs to contain at only information which is relevant for solving the task.

(24) introduced four robotic priors, which were expressed as a loss function, to optimise a state encoder network ϕ. It was hypothesised that all task relevant information can be expressed in a low dimensional state. This hypothesis is not encoded using a loss function, but is implicitly assumed when setting the dimensionality of $\hat{s}_t << o_t$. The priors that were introduced in this work will be discussed below. Other priors have been proposed in literature, however only priors that are relevant for this work will be discussed. Finally, a set of priors, which were designed by the author of this work will be presented.

Originally proposed priors

This section will cover the priors proposed in (24). Each prior will be discussed briefly, and the associated loss function is introduced and analysed.

Temporal coherence prior
This prior assumes that many real world phenomena only change slowly over time. Objects have inertia and therefore only change their velocity slowly as a result of external forces. This prior assumes that other real world aspects to also change gradually. This prior favors state representations that change gradually between observations. The state change is defined as $\Delta\hat{s}_t = \hat{s}_{t+1} - \hat{s}_t$. The temporal coherence loss is defined as:

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

Proportionality Prior
This prior encourages the state representation to change in proportion to the action. This prior results from the Newton's second law of motion. It is assumed that this does not only hold for the robot and interactions of the robot with other objects, but also for more abstract processes. This prior encourages that the same action result in the same change in state. The loss is defined as:

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

Causality Prior
This prior states that if the same action leads to a different reward in different states, these situations must be different in some relevant manner. As such, these situations must not be represented by the same state. It is stated that this prior favors state representations that include task relevant properties to distinguish these different situations (24).

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

Repeatability Prior
This prior states that the consequence of an action should be similar if the situation in which the action is performed is similar. This and the causality prior are inspired by Newtons third law of motion.

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

The computed losses for each of these priors can then be used to optimise a neural network using gradient descent. The total loss is computed using: $L = \omega_1 L_{temp} + \omega_2 L_{caus} + \omega_3 L_{Prop} + \omega_3 L_{rep}$. The weighting parameters ω_x are used to change the relative importance of each of the priors.

The proportionality prior and repeatabiltiy priors defined in 3.2 and 3.4 have the condition $a_{t_1} = a_{t_2}$. The causality prior defined in 3.3 has the condition $a_{t_1} = a_{t_2}, r_{t_1+1} \neq r_{t_2+1}$. These conditions are enforced by finding pairs of training tuples which comply with these conditions. Only pairs of training tuples which comply to the condition of the prior is used to compute the loss on that prior and train the encoder network.

Exploiting the reward landscape for learning the state representation

In the work of (11), the priors proposed by (24) were adapted to better exploit information present in the reward function. Two priors were adapted, the proportionality prior and the repeatability prior. The new priors are:

$$L_{\text{prop}}(D,\phi) = \mathbf{E}\left[\left(\|\Delta\hat{s}_{t_2}\| - \|\Delta\hat{s}_{t_1}\| \right)^2 \middle| |\Delta r_{t_1} - \Delta r_{t_2}| < \kappa \right] \tag{3.5}$$

$$L_{\text{rep}}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2} \|\Delta\hat{s}_{t_2} - \Delta\hat{s}_{t_1}\|^2 \middle| |\Delta r_{t_1} - \Delta r_{t_2}| < \kappa \right] \tag{3.6}$$

Where $\Delta r_t = r_{t+1} - r_t$. As can be seen from the equations, the selection criterion has been changed from equal actions to similar reward variation. This encourages the network to shape the state representation such that pairs with similar reward variation have a similar magnitude change in the state space.

State representation learning for continuous actions space

The priors introduced in (24) were designed with a discrete action space in mind. The work of (36) has introduced state representation learning in the context of continuous actions. In this work the temporal coherence prior is used as defined in 3.1. The proportionality prior is adapted to better fit the continuous action space.

$$L_{\text{prop}}(D,\phi) = \mathbf{E}\left[\left(\|\Delta\hat{s}_t\| - e^{\beta}\|a_t\| \right)^2 \right] \tag{3.7}$$

In their work, it was found that these priors alone were not sufficient to learn a good state representation, and therefore, the landmark prior was introduced. This prior will be further introduced later. Equation 3.7 shows a version of the proportionality prior which works between

consecutive states, whereas the proportionality prior in 3.2 works between all observed states, given these comply with the condition.

The originally proposed priors as introduced by (24), have not been tested for a continuous action space. These cannot be used directly in a continuous action space because there will be very few training tuples where two actions are identical. Therefore, adaptations to these priors are made to make these work in a continuous action space. The conditions of these priors need to be relaxed to find more pairs of training tuples that comply with the conditions in the priors, as actions will rarely be equal in a continuous action space. The equality condition $a_{t_1} = a_{t_2}$ in the priors can be replaced with a similarity condition $\|a_{t_1} - a_{t_2}\| < \epsilon$. The causality prior in 3.3 adds the additional condition $r_{t_1+1} \neq r_{t_2+1}$ to the causality prior. This is designed for the sparse reward setting, where many rewards are equal. In the dense reward setting, even states that are very close in the environment will have different rewards. Therefore, this condition should be changed to $|r_{t_1+1} - r_{t_2+1}| > \mu$ in the dense reward setting. The temporal coherence prior, as defined in equation 3.1 does not have any condition, and therefore does not need to be adapted. The adapted priors then become:

$$L_{\text{prop}}(D,\phi) = \mathbf{E}\left[\left(\|\Delta\hat{s}_{t_2}\| - \|\Delta\hat{s}_{t_1}\|\right)^2 \middle| \|a_{t_1} - a_{t_2}\| < \epsilon\right] \tag{3.8}$$

$$L_{\text{caus}}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2} - \hat{s}_{t_1}\|^2} \middle| \|a_{t_1} - a_{t_2}\| < \epsilon, |r_{t_1+1} - r_{t_2+1}| > \mu\right] \tag{3.9}$$

$$L_{\text{rep}}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2} - \hat{s}_{t_1}\|^2} \|\Delta\hat{s}_{t_2} - \Delta\hat{s}_{t_1}\|^2 \middle| \|a_{t_1} - a_{t_2}\| < \epsilon\right] \tag{3.10}$$

The priors introduced by (11), as defined in equations 3.5 and 3.6 can also be used in the continuous action setting. This can be combined with the causality prior defined in 3.9 and the temporal coherence prior from 3.1.

3.2.2 Removing the conditions from the priors

The above presented priors have been conceived with a discrete action space in mind. These priors use conditions, which match points s_{t_1} and s_{t_2} based on these conditions. While in the discrete action space, the points which are matched based on the actions can be selected for equal actions, in the continuous action space, these have to be selected based on similarity. This introduces additional hyper-parameters which will need to be tuned to obtain a good state representation. These hyper-parameters might also require different values for different environments. This has the downside of introducing additional engineering effort. Furthermore, these hyper-parameters add a discrete condition, while the action space is continuous. Finding points which comply to these conditions also introduce additional computational complexity. To better take advantage of the continuous nature of the action and reward spaces, these conditions can be replaced by a continuous expression which increases or decreases the loss based on the similarity of the actions or rewards in the training pair. These novel priors are the main contribution of this work.

Temporal coherence prior

The temporal coherence prior did not have any conditions attached, only that this prior is used on consecutive states in the sequence. When using continuous actions, the magnitude of the action determines the distance between the two states. When the action taken is large, the temporal coherence loss should be lower than when the taken action is small. This leads to the following prior:

$$L_{\text{temp}}(D,\phi) = \mathbf{E}\left[\left(\|\Delta\hat{s}_t\| e^{-\alpha\|a_t\|}\right)^2\right] \tag{3.11}$$

Here $\alpha$is a hyper-parameter which can be tuned to scale the state space.

Proportionality prior
The proportionality prior encodes the heuristic that the difference in the state space should be similar if the taken action is also similar. In the continuous action space, this could be interpreted such that the difference in the state space between consecutive states should be more similar if a similar action is taken in the state s_t. As the difference between the actions taken from both states is more different, the loss on this prior should be lower.

This lead to the following prior:

$$L_{\text{prop}}(D,\phi) = \mathbf{E}\left[\left(\|\Delta\hat{s}_{t_2}\| - \|\Delta\hat{s}_{t_1}\|\right)^2 e^{-\beta\|a_{t_1}-a_{t_2}\|^2}\right] \tag{3.12}$$

The $\beta$is a hyper-parameter which can be tuned to control how similar the actions should be to get a high loss.

Repeatability prior
This prior should give a high loss when a similar action is performed from a similar state, and the resulting change in the state is not similar. Again, the condition that the actions performed from both states should be similar can be replaced by a continuous component.

$$L_{\text{rep}}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2}\|\Delta\hat{s}_{t_2} - \Delta\hat{s}_{t_1}\|^2 e^{-\beta\|a_{t_1}-a_{t_2}\|^2}\right] \tag{3.13}$$

Causality prior
This prior encodes the heuristic that if the same action results in different rewards in different states, these states should be different. The loss then should be higher if the actions are more similar. Equally, the loss should be higher if the difference in the loss is larger. If the reward is equal, the loss should be zero. This gives the following prior:

$$L_{\text{caus}}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2} e^{-\beta\|a_{t_1}-a_{t_2}\|^2}|r_{t_1+1} - r_{t_2+1}|\right] \tag{3.14}$$

3.2.3 State representation learning in the partially observable setting

State representation learning has mostly been considered in the fully observable setting. In this setting, each observation corresponds uniquely to a single state in the environment. In the partially observable setting, an observation can correspond to several states. To make the problem markovian, recurrent neural networks can be used. Recurrent neural nets have propagation of information across time, allowing the network to learn temporal connections. This allows the state to be identified based on a sequence of observations instead of a single observation. This allows the state representation to be learned for a wider range of environments.

Landmark prior
Where all previous works have assumed a fully observable environment, (36) considered the case of learning the state representation from observations in a partially observable environment. To make the problem markovian, memory was added to the encoder network ϕ by means of an LSTM layer. In their work, the task of navigating a maze was considered. A new prior was introduced, called the landmark prior. This prior uses a set R of l landmarks. For each landmark s_l, all states, for which the ground truth state is within a range of s_l are selected. From these points, k pairs (s_{t_1}, s_{t_2}) are selected. These are selected by finding the k pairs for which the true state is most similar. The states s_{t_1} and s_{t_2} must be from different episodes (trajectories). These k pairs are used to minimize the distance between sequences in the state space. The landmarks are distributed evenly across the environment.

$$L_{\text{landmark}}(D,\phi) = \mathbf{E}\left[\|\hat{s}_{t_1} - \hat{s}_{t_2}\|^2 \middle| s_{t_1} \approx s_{t_2} \approx s_l; s_l \in R\right] \tag{3.15}$$

This Landmark prior makes use of ground truth data to match the underlying states s_{t_1}, s_{t_2} and landmark state s_l. This work will investigate whether the continuous priors introduced

for the fully observable setting are sufficient to learn a good representation when in a partially observable setting.

3.2.4 Hyper parameter analysis

This section will cover the relevant hyper parameters for the state representation learning framework, using the robotic priors presented above. A short analysis of each of the hyper-parameters will be given.

- Matching parameters: These parameters are used in the discrete priors for finding pairs of observations for training.
  - ε: Threshold for setting how similar actions should be. Determines the range of actions on which the prior will work.
  - $\varkappa$: Threshold for how similar reward differences should be for the prior to work on it.
  - μ: Threshold for the minimum difference in reward for the causality prior.
- Prior parameters: These parameters are used in the continuous priors.
  - α: Constant dictating how fast the loss of the temporal coherence prior goes down with increased action magnitude.
  - β: Constant determining how fast the loss decreases with increased difference in actions.
- Prior weighting parameters ω: The weighting parameters which weight up the individual losses from each of the priors can be tuned to change the relative importance of each of the priors.

3.2.5 Reinforcement learning method

The reinforcement algorithm used in this work is DDPG (48). The DDPG algorithm is designed to work with a continuous action space, has a relative good sample efficiency as it is an off-policy method. Furthermore, the DDPG algorithm is easy to implement. These reasons make the DDPG algorithm a good method for this work.

As the DDPG algorithm can get stuck in a local optimum, noise can be added to the actions to aid in exploration. A good exploration of the environment can help avoid local optima. For the purpose of exploration of the environment, Ornstein-Uhlenbeck (OU) noise is added to the actions. This is correlated noise, which helps with better exploration of the environment, versus uncorrelated noise, as uncorrelated noise has zero mean.

The DDPG algorithm has several hyper-parameters which need to be tuned. These parameters need to be tuned such that the algorithm works well and reliably. These hyper-parameters will be introduced and explained further.

Neural network architecture

The neural network architecture of both the actor and the critic can be of crucial importance to the performance of the algorithm. While many different architectures could be used, the fact that in this work the input is a low dimensional state makes the design easier. Given that the input is only a low dimensional state, using only fully connected layers is sufficient. To further simplify the design, the network architectures of the actor and the critic can be set to be almost the same, with only the output layer being different. The output layers of the actor and the critic needs to be different, as the critic only has one output value, the Q value, and the actor has as many outputs as the number of different outputs.

Learning rate
The learning rate α is of high importance to the performance of any neural network. The learning rate which gives good performance depends on a number of factors. These factors include the neural network architecture, the batch size with with the neural network is trained and the optimization method with which the gradients are computed. The learning rate needs to be set experimentally as no analytical methods are available for computing the learning rate. The learning rate can be different for the actor and critic networks.

Discount rate
The discount rate γ determines how much the algorithm values the rewards in the current step versus the rewards in future steps. This parameter controls how greedy the policy will be with regards to obtaining rewards.

Batch size
The batch size is the number of points, which are sampled from the replay memory at random, which is used for updating the network. The batch size influences the stability of the learning, as well as the speed.

Limit on actions
The actions which can be taken by the actor network are limited to set parameters. The limit can be different for each output of the actor network, as these outputs control separate properties of the robot.

Exploration policy
To aid with exploration, OU noise is added to the action, i.e. to the output of the actor network. This sum of action and noise is then limited to the same limit as which the actions are limited to the same limit as the output of the actor network.

Reward function
The reward function can be set to different functions. The reward can be based on different metrics, a common metric is the euclidean distance between the robot and the target. The reward function can be non-linear, an exponential function is commonly used to accelerate the learning process.

Maximum number of steps
There is a maximum number of steps that can be taken within one episode. If the robot has not collided with any wall or reached the target, the episode will end, and the environment will be reset to the begin state.

The DDPG algorithm has the state as input. This state is computed by the state representation network such that $\hat{s}_t = \phi(o_t)$. The learned state should contain all information relevant for solving the task, and should be unambiguous, as ambiguity will hamper the learning of a good policy.

The actor and critic networks will be updated once for each time-step taken. The updates will take place after an episode has ended. An episode ends once the agent has reached the target, hit a wall or has reached the maximum number of steps for that episode.

3.2.6 Mobile robot analysis

The experiments for answering the research questions will be performed in a simulation environment. The mobile robot of choice is the Turtlebot3 waffle pi (4), as shown in Figure 3.1

The Turtlebot mobile robot is chosen for several reasons. The first reason is that this mobile robot features both a lidar and a camera sensor, which combined is a very versatile package. These sensors can be used for many different tasks, and are well suited for the task of mobile robot navigation. Furthermore, these sensors are both high dimensional, making these suitable for state representation learning. The Turtlebot3 is relatively affordable, making real world

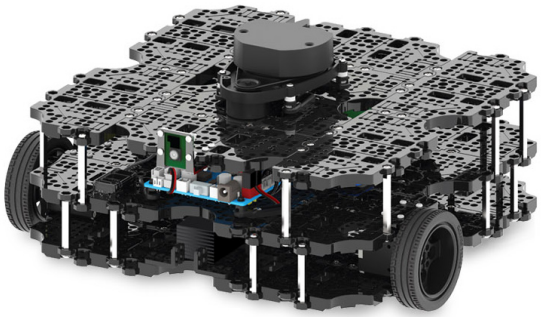


Figure 3.1: The Turtlebot3 waffle pi mobile robot

experimentation or validation feasible. The Turtlebot3 is based on the ROS platform, and has open source software available for communication and simulation. This makes the use and implementation of this robot much easier. A model for the Gazebo simulation was already available can be used unchanged in this work. Therefore, the Gazebo simulation environment will be used for experimentation.

Robot control

The robot has two driven wheels to control the robot. Steering is done using differential drive. Each wheel is controlled using a DYNAMIXEL actuator (3). The Turtlebot robot is controlled using velocity commands. These commands consist of a command for the translational velocity and the rotational velocity. An internal controller then translates these velocity commands to obtain the required velocities for each of the wheels, such that the required translational and rotational velocity is achieved. This internal controller can sent the required velocities directly to each wheel, as the DYNAMIXEL actuator has an internal controller to control the motor speed. An overview of this control scheme can be seen Figure 3.2.

Instead of sending commands for the required translational and rotational velocities, one could also directly control the speed of the wheels. This requires the reinforcement learning algorithm to learn the kinematics of the robot. Although the kinematics are relatively simple, this work is not focused on learning robot kinematics. As the internal controller can be used directly without any further engineering cost, this is preferred.

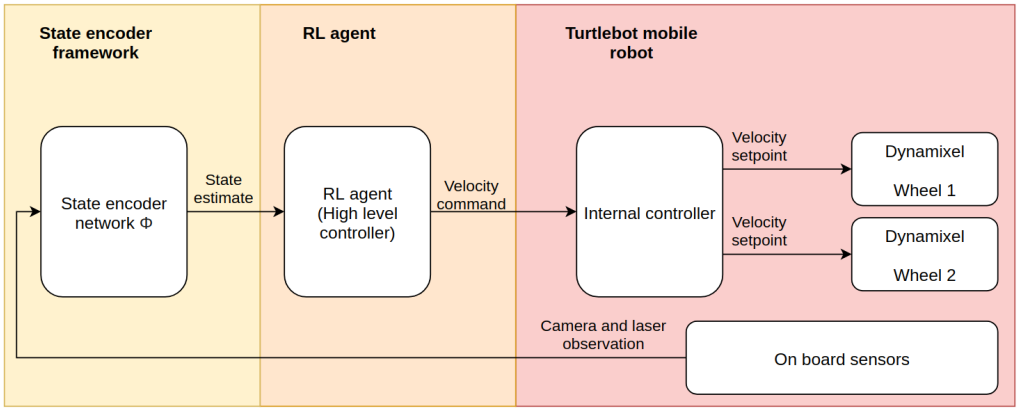


Figure 3.2: Overview of the control structure

Sensor analysis

The Turtlebot waffle pi has two sensors which will be used for learning the state representation. The first sensor which is used is a lidar sensor. The second is a RGB camera.

The lidar sensor on the Turtlebot robot measures the distance between itself and obstacles. There are two main elements to consider:

Range
The range of the laser dictates the maximum distance at which obstacles can be detected. This is mainly dictated by limitations in the sensor, although this limit can be artificially lowered.

Lidar samples
The lidar sensor works in 360 degrees. The number of lidar samples is the number of measurements taken in these 360 degrees. Having more measurements gives more information about the shape and distance of obstacles. More measurements also means higher dimensional data.

The Turtlebot3 waffle pi makes use of the raspberry pi camera. The two main components of this camera are:

Resolution
The resolution of the camera can be important for learning the state representation. A higher resolution will allow for greater detail in the image. However, the higher resolution also means a higher computational cost to run it through the network. A higher resolution image also requires a more complex convolutional network to give the model increased capacity for exploiting these details.

Field of view
The field of view is an important aspect of the camera, especially in the context of state representation learning. The field of view determines how much of the environment can be seen in one observation. If more of the environment can be seen in one observation. This makes learning a state representation easier.

3.3 Methodology

This work will be focused on two parts. The first part of this work will focus on learning the state representation in a continuous action space. The second part of this work will be focused on learning a state representation in a partially observable setting.

To perform the experiments, a simulation environment is used. This simulation environment should mimic the real world. The Turtlebot mobile robot should be implemented into the simulation environment. The simulation environment should allow for easy interfacing with the implemented state representation framework and RL algorithm.

The work will be performed in a simulation environment, which will model real world behaviour of the mobile robot. The robot has a camera and a lidar sensor, as mentioned previously. The model will be able to simulate the Turtlebot's kinematics and dynamics. The simulation is implemented using Gazebo, an open-source robotics simulation platform. Gazebo interfaces with ROS (5). As Gazebo is an established platform, there is open source code available which enables this interaction. The interface with ROS is used to control the robot and retrieve observations from the robot. This can be done using OpenAI's gym (40) and python. This interface had already been set up, and had only to be adapted for this work. Many of the visualisation tool had already been implemented, and could be used as-is.

The code for learning the state representation and the reinforcement learning algorithm is written in python. Python has a wide choice of packages for machine learning purposes available. The package used in this work for both the state representation learning method and the reinforcement learning algorithm are build using Tensorflow (6). The DDGP algorithm has been implemented from scratch for this work. The state representation method was already available at the start of this work, but was made to work for a discrete action space. This code had to be modified extensively to work for the purposes of this work. Further work had to be done to

implement a recurrent neural network, and integrate this into the state representation learning framework.

For answering the research questions, the proposed state representation framework using the robotics priors should be tested across different environments, as this gives a more robust evaluation of the proposed robotic priors. To evaluate the performance of the different sets of robotics priors for learning the state representation, several evaluation metrics can be used. These evaluation metrics will be used when these give more insight into the performance of the state representation learning.

3.3.1 Evaluation metrics

The evaluation metrics that can be used for evaluating the performance of the state representation framework using the proposed robotic priors will laid out in this section.

Cumulative reward
The cumulative or episode reward is the sum of all rewards the agent receives throughout one episode. The goal of learning the state representation is ultimately that the reinforcement learning algorithm is able to learn a good policy based on this representation. Therefore, the quality of the learned policy is of great importance. One way to judge and compare the quality of the learned policies is to look at the cumulative reward. The higher this cumulative reward, the better the policy, as reaching the target in less steps will result in a higher cumulative reward.

Terminal reward
The terminal reward is the last reward received in an episode. The terminal reward will show whether one of the walls was hit, the target reached, or if the episode terminated without either. The terminal reward will thus show how reliably the robot reaches the target location and when the robot crashes into a wall. If the representation is good, the policy should learn to reach the target location reliably.

Smoothness of the representation
The learned state representation can be judged by looking at the smoothness of the learned representation. This can be achieved by plotting the learned representation versus the rewards. The reward function is a smooth function throughout the environment, and only is discrete when a wall is hit or the target reached. Therefore, the reward should change smoothly throughout the environment, and thus also throughout the learned state representation.

Correlation analysis
The state representation network ϕ should learn to encode relevant information for solving the task. Solving a navigation task towards a stationary target can be achieved using only information on the position and orientation of the mobile robot. A good representation should learn to encode this basic information. Therefore, the correlation coefficient between the ground truth position and orientation and the learned state representation can be used to evaluate whether the state representation network has learned to encode this information.

3.4 Summary

In this chapter, the research questions which have been the starting point of this work were introduced. The first research question: *How can robotic priors be used to learn a state representation in case of a continuous action space?*. And the second research question: *To what extend can a recurrent state representation be learned using the adapted priors in a continuous action space?*

After introducing the research questions, an analysis of the various systems was presented. Firstly, this covered an analysis of the state representation framework. Here, an analysis is made

of the different method of learning a state representation. This analysis concludes with the reasons for learning a state representation using robotic priors.

The robotic priors, which are encoded as loss functions, can take many shapes. Several of these priors have been presented in literature. The priors which are relevant for this work are introduced and analysed in this section. The robotic priors which have been designed for this work are also introduced here. After introducing these novel robotic priors, the partially observable setting is considered. An analysis is given of the work in literature which have considered the partially observable setting while using robotic priors to learn a state representation.

An analysis is made of the reinforcement learning algorithm. This covers an analysis of the parameters and hyper-parameters which will need to be set, and will affect the performance of the algorithm. The choice of reinforcement learning algorithm is also analysed. The final part of the system that was discussed is the mobile robot. The choice of mobile robot is considered, as well as some of the aspects of the robot that are important. This includes the control of the robot, as well as an analysis of the sensors and its properties.

Finally, the methodology for this research is laid out. This section has covered the tools that are used to do this research, and has laid out the way in which the performance of the methods will be evaluated. The main metrics of evaluation are the cumulative reward, the terminal reward and the more qualitative metric of smoothness of the learned representation.

4 Design and implementation

This chapter will discuss the design and implementation of the experiments and the frameworks presented in the previous chapter. First, the design of the experiments will be covered, which will cover which experiments will be performed for answering the research questions. Then the implementation of the RL algorithm and the state representation learning framework is expanded upon. The chapter will then further elaborate on the simulation setup.

4.1 Experimental design

In this section, we will touch upon the setup of the experiments. This section will cover the different environments which have been used to perform the experiments, as well as the structure and goals of the experiments. This section will first discuss the experiments used for answering research question 1, and then go on to cover the experiments for research question 2.

4.1.1 Research question 1

This section will cover the experiments that will be performed for answering **RQ1**. This first research question is repeated here:

RQ 1 How can robotic priors be used to learn a good state representation with a continuous action space?

This question is split into two sub-questions. The first sub-questions:

1a Are the proposed priors from (24) and (11) able to learn a good representation when using a continuous action space, when these are adapted to better work with continuous actions?

The adaptations made to the priors from (24) have been presented in the previous chapter. The previous chapter has also covered the adaptations made to these priors to better exploit the reward signal, which were proposed by (11). These adaptations can also be combined with the priors adapted to work with continuous actions. To answer the research question, and also consider the adaptations proposed by (11) two sets of priors are made.

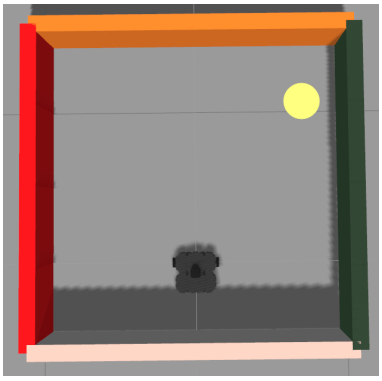
Prior set one

The first set of priors to be tested will consider the priors which were adapted from the work of (24). This set consists of the temporal coherence prior as defined in 3.1, the causality prior as defined in 3.9, the proportionality prior as defined in 3.8 and the repeatablity prior as given in 3.10.

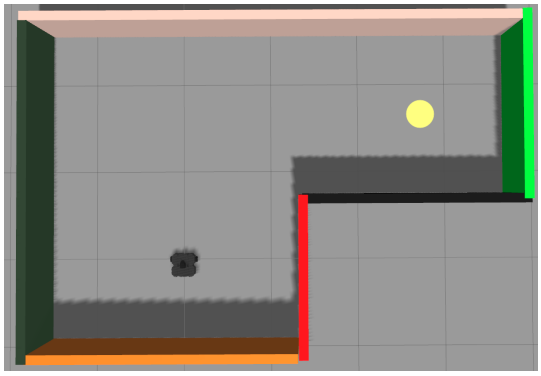
Prior set two

(11) has changed the proportionality and repeatablity priors to better exploit the reward signal. These priors used the equality condition $a_{t_1} = a_{t_2}$, this condition is changed to $|\Delta r_{t_1} - \Delta r_{t_2}| < \kappa$. These new priors can be found in equations 3.5 and 3.6. These priors can be combined with the temporal coherence prior and adapted causality prior from *Prior set one*, to create a second set of priors.

1b Can the current sets of priors be adapted to better make use of the continuous nature of both the reward signal and action space by removing conditioning on the training points?



(a) Overview of the small square environment



(b) Overview of the L-shaped environment

Figure 4.1: Overview of environments showing the initial position of the robot and the target location with the yellow circle

The manner in which the priors are adapted to remove the conditions from the priors and make them continuous is covered in section 3.2.1. This gives two extra sets of priors to be evaluated.

Prior set three

The priors of set three are adaptations of the priors in set one. The temporal coherence priors, causality prior, proportionality prior and repeatebility prior can be found in equations 3.11, 3.14, 3.12, and 3.13 respectively.

Prior set four

In fourth and final set of priors, the term $e^{-\beta\|a_{t_1}-a_{t_2}\|}$ is replaced with the term $e^{-\beta\|\Delta r_{t_1}-\Delta r_{t_2}\|}$ in the proportionality prior and repeatabilty prior from equations 3.12 and 3.13. This makes the fourth set of priors the continuous equivalent of the priors in set two.

These four sets of priors will be tested in the environments which will be presented below. The performance of these different sets of priors will be evaluated in these environments, based on the evaluation metrics introduced in 3.3.

Environments

To test the priors different sets of priors, the environments in which these are tested must be sufficiently diverse in layout and difficulty. Another consideration with a continuous action space is that the scale of the actions can be different between different environments, in a large environment one would typically increase the action limit. The action limit can impact the learned representation, and the robotic priors should be able to learn a good representation even when the agent is allowed to take larger actions.

The environments that are used for answering **RQ1** are covered here. The first environment that is used is a small square. Each of the four walls have a different color, as there is no texture, the different colors of the walls make the environment fully observable.

Figure 4.1a shows a top view of the environment. The yellow circle indicates the target location, this is added for illustration purposes and cannot be seen in the real environment. The Turtlebot3 is shown in its initial position. This is true for all environments that are shown.

This environment small square is similar to the environment used in (24). This environment is relatively easy, as the agent needs few steps to reach the target.

The second environment is a bit more complex, this environment is called the L-shape environment. This environment will further test the priors ability to learn a good representation. This is especially true because the area near the target location will be less explored than the

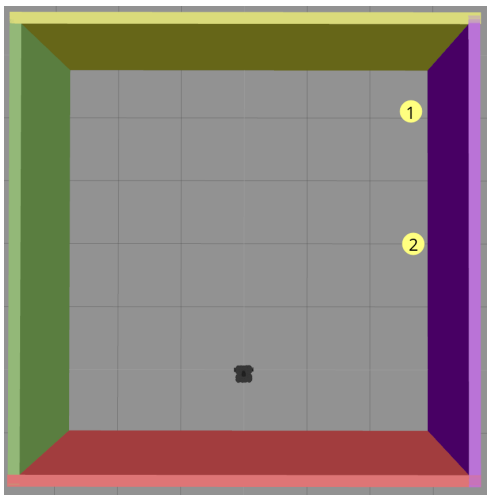


Figure 4.2: Overview of the large square environment showing the initial position of the robot and the two target locations

rest of the environment when using a random policy to gather the data, as is done to gather training data. Therefore, the data which is gathered in this environment will not be uniformly distributed across the environment. This will pose a challenge, as even the less dense area need to be represented such that a good policy can be learned. Figure 4.1b shows a top view of the environment, as well as the starting position of the agent and the target location.

The third environment is again a square environment with differently colored walls. This environment however is much larger. This environment is designed to be such that when close to the walls, no corner can be seen in the camera image. Given the much larger size of this environment, the translational velocity limit is increased in this environment. This will give an indication of how well the priors can handle the different action magnitudes.

This large square environment is shown in Figure 4.2. In this environment, two target locations can be seen. Target location one is positioned close to one of the corners, and is used in the experiments of research question 1. The second location is positioned somewhere near the middle of one of the walls, this target location is used exclusively for experiments in research question 2.

Each set of priors will be tested on these environments. First these will be tested on the small square environment, then these will be tested on the L-shaped environment. Finally, these priors are tested in the large environment and tested for the ability of RL algorithm to learn a good policy.

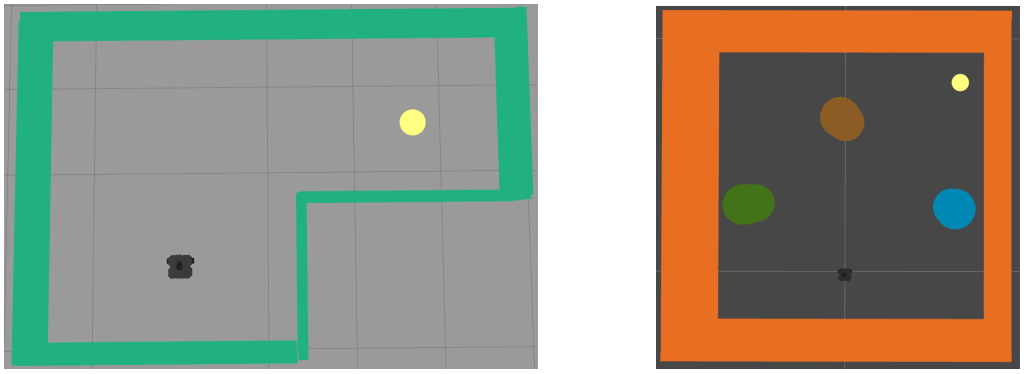
4.1.2 Research question 2

The second research questions considers more challenging environments. To learn a state representation in these environments, a recurrent state representation encoder will be used. For each environment, a comparison will be made between the performance of the non-recurrent state representation encoder network and the recurrent version. To train the recurrent encoder network, the priors from set three will be used. The second research question is:

RQ 2 To what extend can a recurrent state representation be learned using the adapted priors in a continuous action space?

Environments used for answering RQ 2

To test how well these priors can learn a state representation when the encoder network is recurrent, more challenging environments need to be created. These environments will be made



(a) Overview showing the L-shaped environment with one color wall

(b) Overview of the large square environment with colored cones

Figure 4.3: Overview of environments used for research question 2, showing the initial position of the robot and the target location

such that the robot cannot be uniquely localised by looking a the color of the walls. This makes learning a good state representation more challenging, and in some states partially observable.

The large square environment, seen in Figure 4.2 was created such that in some states, the colors of the walls can be used to uniquely define the state. In other states, mainly when close to one of the walls, only one wall can be seen in the camera image, such that the state cannot be uniquely determined by the camera image. This gives that the location of some targets, like target location one, are much easier to encode well. Target location two on the other hand is much harder to encode, as the camera image is not sufficient to determine a unique state of the mobile robot. This environment should allow for a good comparison between the use of a recurrent model and the non recurrent setting. This can be done by evaluating the learned representations, and making comparisons between those. Then a comparison can be made between the learned policies.

To further evaluate the effectiveness and the limits of the recurrent model, more complex environments are introduced. The L-shaped environment from Figure 4.1b will be adapted. Instead of differently colored walls, all walls will have the same color. Also, the shadows will be removed to make each wall the same shade. The environment can be seen in Figure **??**. This environment is complex because the color of the walls cannot be used to determine the position. Instead, the state representation network must rely on the shape of the walls and on the lidar sensor. Using a recurrent state representation network can help to disambiguate the states, especially when the observations can be very similar in different states.

A third environment for evaluating the recurrent state representation framework is shown in Figure 4.3b. This environment has the same dimensions as the large square environment. The walls are all the same color and symmetric, making the environment partially observable. Three colored blocks are added into the environment. These colored blocks make some states, the states in which one of these blocks can be seen, fully observable. However, to get to one of the corners, there is a long sequence of non-unique states, making it difficult for the recurrent state representation framework to find a good representation, and requiring a long sequence length during training.

4.2 Algorithm Implementation

The different components of the overall framework are covered in the previous chapter. This chapter will look at the elements from the reinforcement learning algorithm and the state representation learning algorithm. This chapter will cover further implementation details of these algorithms.

4.2.1 Implementation of DDPG

The DDPG algorithm was chosen as the algorithm for the reinforcement learning part of this work. The technical details of how this algorithm works is covered in section 2.1.

As mentioned previously, the DDPG algorithm was designed for a continuous action space. The DDPG algorithm is an actor-critic method. Actor-critic methods have two separate networks, one network for estimating the action-value function, or Q-function. This is called the critic network. The actor network outputs the action. In discrete action spaces, the critic network can have multiple outputs, it has one output for each of the different actions. In a continuous action space, this would be impossible, given the size of the action space. To make the Q-function practical to implement, this function has only one output value. Instead of giving an output for each possible action, the output of the actor network a_t is concatenated with the state $\hat{s}_t$, such that the input dimensionality of the critic network is the sum of the dimensionality of the state and the action space. Note that as the state representation network estimates the state from observations, and the state is then fed to the RL algorithm. The state $\hat{s}_t$ is not the true state, but instead the estimated state, such that $\hat{s}_t = \phi(o_t)$ with ϕ the state representation encoder network.

The actor has only the state as input. In our specific implementation, the dimensionality of the action space is two, one for the desired translational velocity, and one for the desired rotational velocity. In our experiments, the actions are limited to maximum and minimum values. The translational velocity is limited between zero and ζ. This means that the robot is not able to go backwards. The upper limit ζ is set depending on the size of the environment, where larger environments have a higher upper limit. The rotational velocity is limited between -0.2 and 0.2 in all experiments. This gives the robot sufficient maneuverability to reach each point in the environments. As the output for the translational velocity is clipped to be equal or larger than zero, and the rotational velocity can be smaller than zero, the two outputs need different output activation functions. For the translational velocity output, the sigmoid activation function is used, and for the rotational velocity output the tanh activation function is used.

As described in the background section, target networks are used in the target estimation in the bellman equation, as this makes learning more stable. Both the actor and critic have a target network. The DDPG algorithm uses soft target network updates, meaning that the values of the target networks are updated using polyak averaging. The target networks are updated at each training step, so each time the main networks are trained. This is done according to $\theta_{\text{target}} = \rho\theta_{\text{target}} + (1-\rho)\theta$. Here ρ is a parameter between zero and one which can be tuned. A low value gives higher training speed, while a high value will give a more stable learning process.

The actor and the critic networks can be updated separately. Each learning step, the critic network is trained first, then the actor network is trained. The critic network is updated according to the bellman equation, where the bellman equation is the target. The full loss function is given in equation 2.11. The actor is instead updated by optimising the Q-value for the action. The gradient for the actor is computed using equation 2.12.

The DDPG algorithm is an off-policy algorithm. This means that the networks can be trained using data which is collected from a different policy. This means that a replay memory can be used for training a policy. A replay memory stores past experiences. At each training step, a batch of samples is pulled from this replay memory at random. This ensures that the samples in the batch are not temporally correlated, which aids in training. The size of the batch can be set and tuned for good performance.

The size of the replay memory is also an important aspect to tune. Increasing the size of the replay memory can prevent catastrophic forgetting, a phenomenon in neural networks where the neural network will forget previously learned information when trained with new information.

In the setting of the state representation learning framework, the replay memory should not be too large. When the state representation network is trained, the states which were stored in the replay memory of the DDPG algorithm are no longer of the same distribution as the states which are encountered in the environment. Having a relatively small replay memory allows these states from the old distribution to be replaced by the new states before the state representation is trained again.

Exploration policy

Exploration of the environment is very important in the RL setting. Having a good exploration policy will aid in finding a good policy. A lack of exploration often leads to sub-optimal performance. The easiest way to aid exploration is to add noise to the actions which are taken by the actor. The DDPG algorithm is deterministic, such that when no noise is added, the actor will perform the same actions is the same states each time, given that the network is not trained in between. Adding noise is crucial to ensure sufficient diversity in the training data, helping with finding a better policy. The authors of the original DDPG paper (48) advised adding Ornstein-Uhlenbeck (OU) noise (50), which is time correlated noise. This was therefore used. As advised by OpenAi (2), the scale of the noise is reduced over the course of training. As training restarts when the state representation is trained, due to the fact that the distribution of states changes, the scale of the noise is reset when the state representation network is trained. It was further found that when the noise level is kept constant, the policy learns to partially correct for the noise, which deters the algorithm to explore the environment.

Exploration is very important in the normal RL setting, but when using state representation learning using robotic priors, exploration is even more crucial. A good exploration of the environment allows the state representation network to learn a good representation, which is again important for finding a good policy, as the policy is based on the learned state representation. As the samples need to be sufficiently distributed across the environment to learn a good state representation, the magnitude of noise which is added is kept relatively large.

The OU noise can be estimated by using the following equation, as implemented in the openAI baselines (39):

$$x_{n+1} = x_n + \theta(\mu - x_n)\Delta t + \sigma\sqrt{\Delta t}\mathcal{N}(0,1) \tag{4.1}$$

Practical implementation

The task is of episodic nature, an episode starts in the same position and rotation. The estimated state at this point is $\hat{s}_0$. Each episode has a maximum number of time steps before it terminates, which is set to 150. For more complex environments this could be set higher. Each episode will terminate on one of the three conditions. The first condition is that the robot has hit one of the walls, after which the robot receives a negative reward. The second is that the goal has been reached, then the agent receives a positive reward. Finally, the episode can end if neither of these conditions has been met, but the maximum number of steps have been taken.

At each time step, the observation from both the lidar sensor and the camera, which together is o_t, is used to estimate the state $\hat{s}_t = \phi(o_t)$. Then $\hat{s}_t$ is used by the actor, or policy network π to compute the action $a_t = \pi(\hat{s}_t)$. Noise is added to this action, $a_t = \pi(\hat{s}_t) + \epsilon$. This action is then sent to the controller, to execute the action. After the action is performed, the environment will sent the new observation o_{t+1}, the reward for being in that new true state r_{t+1} and a binary value indicating whether this new state is a terminal state or not. This information will be saved to the replay memory for training the network. The information is saved in tuples, with $(\hat{s}_t, a_t, r_{t+1}, \hat{s}_{t+1}, d)$ where d is the binary value indicating if $\hat{s}_{t+1}$ is a terminal state.

The Q-value network and the policy network will be trained with a mini-batch for each time-step. This is done at the end of an episode. The Q-value network is updated according to the loss function defined in 2.11. If the training state $\hat{s}_t$ is a terminal state, the Q value for $\hat{s}_{t+1}$

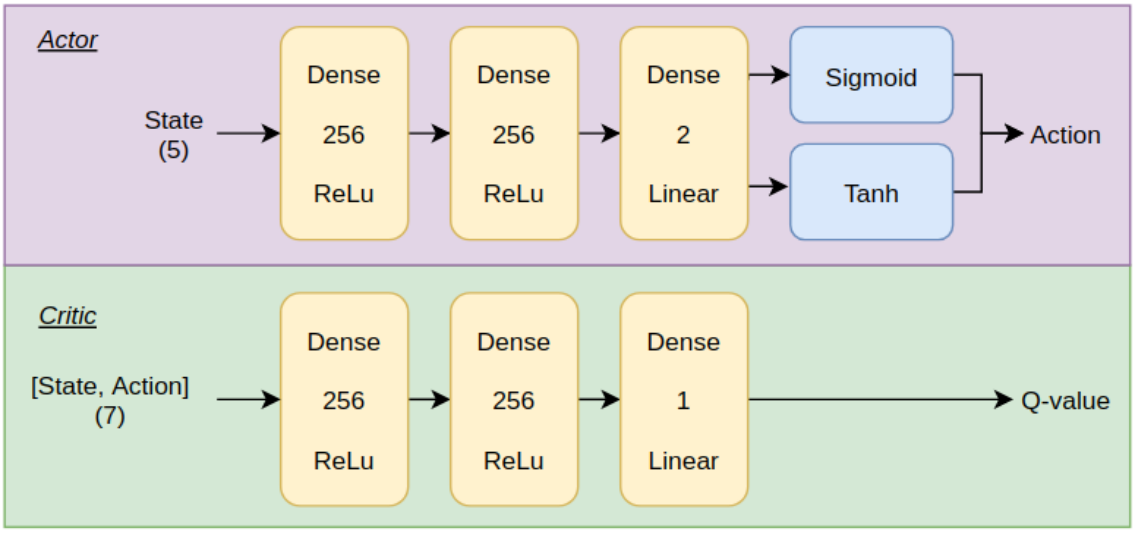


Figure 4.4: Reinforcement learning neural network architecture overview

should be zero. Exploiting the fact that $d = 1$ if $\hat{s}_t$ is terminal, and zero otherwise, the loss function can be implemented as:

$$L_{i_{\text{critic}}}(\theta) = \mathbb{E}_{;}\left[\left(r_{t+1} + \gamma(1-d)Q(\hat{s}_{t+1}, \mu(\hat{s}_{t+1};\theta'_{\mu});\theta'_{Q}) - Q(\hat{s}_t, a_t;\theta_Q)\right)^2\right] \tag{4.2}$$

Neural network architecture

The architecture of the neural network can be an important part of the success of a reinforcement learning algorithm. The network should be sufficiently expressive to effectively represent the Q-value function as well as a good policy. The actor and critic networks are kept mostly the same, for the sake of simplicity. As the performance of the RL algorithm is not the main aim of this work, the network architecture was not highly optimised.

The dimensionality of the state representation is set to five in all experiments. This means that the actor has five inputs, while the critic has an input dimensionality of seven. Given that the dimensionality of the input space is relatively low, using only fully connected layers in the neural network is sufficient. The network consists of two hidden layers, and an output layer. The number of hidden nodes in these layers is chosen to be 256. It was found that using 128 hidden nodes per layer gave inferior performance, using 256 worked well, and allowed the agent to learn a good policy when using the ground truth state in each of the environments. The activation function used in these hidden layers is the ReLu activation function.

As the robot is limited to a translational velocity larger than zero, while the rotational velocity is limited between -0.2 and 0.2, means that both outputs need a different activation function. The output for the translational velocity has a sigmoid activation function, which limits the output between zero and one. This output can be multiplied with the action limit to obtain the action. Then the output for the rotational velocity has the activation function tanh. which gives an output between -1 and 1. Again, this output is multiplied with the action limit for the rotational velocity to obtain the action. An overview of the neural networks used for the actor and critic can be seen in Figure 4.4.

Reward function

To aid with learning a good policy, a dense reward function is used. The reward function is designed to aid the algorithm to be able to reach the target quickly. The used reward function is based on the euclidean distance between the robot and the target, with the reward increasing as the distance to the target decreases. To further enhance the speed of learning a good policy, an

exponential factor is used in the reward function. The exponential factor increases the reward difference between states as the distance to the target is larger. This will encourage the agent to move towards the target quicker.

The reward function is continuous within the environment, except for when the robot hits a wall or reaches the target. If the robot hits a wall it will receive reward $-r_t$, and receive reward r_t if it reaches the target. The reward function is defined as:

$$r = \begin{cases} -r_t & \text{if hit a wall} \\ r_t & \text{if reached target} \\ 1 - e^{a*\text{distance}} & \text{otherwise} \end{cases} \tag{4.3}$$

Equation 4.3 shows the reward function which is used in each environment. The terminal reward term r_t can be changed depending on the environment. In our experiments, these were set to $r_t = 20$ for the small square environment and the L-shape environment. For the large square environment, $r_t = 50$. This value was increased, as the cumulative reward is much larger in the large square environment given that is takes more steps to reach the target location. The variable a in the exponential can be used to scale the reward function across the environment. This helps to make the range of the reward similar between environments. This value is different for each environment:

- Small square environment: $a = 0.5$
- L-shape environment: $a = 0.2$
- Large square environment: $a = 0.18$

Used hyper-parameters

This section will cover the hyper parameters used in the experiments. These include previously covered parameters, as well as other parameters that have not been introduced explicitly.

- Discount rate (γ): 0.99
- Batch size: 64
- Replay memory size: 5000
- Maximum steps per episode: 150
- Polyak averaging parameter (ρ): 0.995
- Noise parameters: OU noise is added to the actions during training to aid exploration. The magnitude of the noise is linearly decreased from the highest to lowest magnitude.
  - σ: $\sigma_{\text{min}} = 0.02$, $\sigma_{\text{max}} = 0.15$
  - θ: 0.35
  - Δt: 1
  - μ: 0
  - Decay period, the number of episodes in which σ goes from σ_{max} to σ_{min}: 250
- learning rate Q network (α_{Q}): 0.001
- learning rate actor network (α_{π}): 0.0001

4.2.2 Implementation of the state representation framework

We then consider the design and implementation of the state representation framework. The goal is to learn a state representation which encapsulates all task relevant information and discards information not relevant for the task. The state representation mapping from observation to state $\hat{s}_t = \phi(o_t)$ is implemented using a neural network. The network is optimised using the robotic priors, as presented in the previous chapter.

To compute the loss and train the network, experiences will need to be saved into a memory. The memory will be filled with experiences from the environment, the states will be saved in tuples of (o_t, a_t, r_{t+1}, d) to the state representation memory. In this setting, d is the binary value indicating whether the saved sample is the last sample of the episode. This is slightly different than the RL setting, where d indicated whether it was a terminal state, so only when the agent has hit the wall or reached the target.

Finding pairs of observations

The previous chapter has discussed several pairs of robotic priors which will be tested and compared for this work. The priors which were taken from literature require pairs of observations which are checked for some condition. The original work of (24), finding pairs of training tuples by first sampling a batch of tuples from the memory. Then all combinations of tuples in this batch are evaluated on the conditions. Finally, the loss is computed using all these combinations which comply to the conditions of each prior, after which the state representation network is trained.

Algorithm 2: Training procedure in the work of (24)

```
Sample minibatch of N random indexes I from SRL memory indexes
Initialise empty arrays M_1, M_2 for the matched points
for i in I do
    for j in I do
        if a_i == a_j then
            Append index pair (i, j) to M_1
            if r_i ≠ r_j then
                Append index pair (i, j) to M_2
            end
        end
    end
end
Compute L_temp using samples in I
Compute L_prop and L_rep using the pairs in M_1
Compute L_caus using the pairs in M_2
Compute L_priors = ω_1 L_temp + ω_2 L_caus + ω_3 L_prop + ω_4 L_rep
Optimise φ using L_priors
```

In a discrete action setting, this works well, as there will be many pairs of training tuples with equal actions. In the continuous action setting, this procedure does not work as well, since much fewer pairs of observations will be found to comply to the condition. Although the conditions in these priors are relaxed, this will still be the case. One can vary the threshold variables ϵ and μ of equations 3.9, 3.10, and 3.8 to find more matching points, but this can also have a detrimental effect on the learned representation. Furthermore, this method of finding matching points gives that the number of points used for each of the priors can vary depending on the number of matching observations that are found. Furthermore, it requires evaluating of all pairs of observations, which becomes a very large number of evaluations when the batch size becomes larger. The temporal coherence prior defined in 3.1 does not have a condition. This

means that the number of observations used for this priors is equal to the batch size. If the batch size is increased, the number of matching pairs for the other priors will increase more than the increase in batch size, as the number of possible combinations grows quadratically. Therefore, changing the batch size will change the ratio between the number of training samples used for the computing temporal coherence loss and the number of samples used for the other prior losses. This may require re-tuning the ratios ω_x by which the losses of each of the priors are multiplied in order to get a state representation of similar quality each time the batch size is changed.

For these reasons, the implementation of finding these matching points is changed. Instead of evaluating each possible pair of observations within a sampled batch, only one matching observation is sampled from the state representation memory for each observation in the sampled batch. This makes that each prior is evaluated using an equal number of samples, independently of the batch size.

To find matching pairs efficiently in the state representation memory, one should not perform an exhaustive search. Instead one could use a while loop, and continue evaluating the observation in the sampled batch with on from memory until an observation complying with the condition has been found. However, it was found that in python it was more efficient to compute this condition for multiple points at the time and randomly pick one observation from the observations that comply with the condition. This is more efficient, as this makes better use of the efficient implementations of the python library Numpy. It was found that evaluating 1000 tuples from memory was a good compromise. To ensure a good random sampling over the entire memory, the evaluation starts at a random point in memory, and continue to go through the entire memory.

The priors defined in 3.8, 3.9, and 3.10 each have the matching criteria $|a_{t_1} - a_{t_2}| < \epsilon$, where the causality prior has an additional matching criterion. The other set of priors with matching criteria, of which the causality prior is the same, has the criterion $|\Delta r_{t_1} - \Delta r_{t_2}| < \epsilon$ for the proportionality prior 3.5 and repeatability prior 3.6. As the first set has the same criterion for the three priors, these matching samples can be easily found using the same loop. The procedure for finding the training pairs is detailed in algorithm3 When using the second set of priors, the repeatablity and proportionality prior samples are selected based on a different condition

than the causality prior samples. Therefore, selection of training points for these priors is done according to algorithm 4. The entire procedure is detailed in algorithm **??**.

Algorithm 3: Match points according to criterion 1

Sample minibatch of N random indexes I from SRL memory indexes
Split the SRL memory into $memorysize/splitsize$ splits S
Initialise empty arrays M_1, M_2 for the matched points
for i *in* I **do**
  Select random split S_{start} to start in the dataset, to randomize the matching
  for x *in* S **do**
    $S_y = S_x \bmod S_{\text{start}}$
    Compute the norm used in the first condition using actions in S_y, a_{S_y}, and action a_i: $\|a_{S_y} - a_i\|$
    Find indexes of points which satisfy condition 1: $C_1 = Where(\|a_{S_y} - a_i\| < \epsilon)$
    Compute the norm used in the second condition using the rewards: $\|r_{C_1} - r_i\|$
    Find indexes of points which satisfy condition 2: $C_2 = Where(\|r_{C_1} - r_i\| > \mu)$
    if $len(C_2) > 0$ **then**
      Pick one index c_1 randomly from C_1
      Append c_1 to M_1
      Pick one index c_2 randomly from C_2
      Append c_2 to M_2
      Break out of the for loop, move on to $i+1$
    end
  end
end

Algorithm 4: Match points according to criterion 2

Sample minibatch of N random indexes I from SRL memory indexes
Split the SRL memory into $memorysize/splitsize$ splits S
Initialise empty array M for the matched points
for i *in* I **do**
  Select random split S_{start} to start in the dataset, to randomize the matching
  for x *in* S **do**
    $S_y = S_x \bmod S_{\text{start}}$
    Compute the reward delta for points in S_y: $\Delta r_y = \|r_{S_y+1} - r_{S_y}\|$
    Compute the reward delta for the minibatch tuple: $\Delta r_i = \|r_{i+1} - r_i\|$
    Find indexes of points which satisfy the condition: $C = Where(\|\Delta r_y - \Delta r_i\| < \kappa)$
    if $len(C) > 0$ **then**
      Pick one index c randomly from C_1
      Append c to M
      Break out of the for loop, move on to $i+1$
    end
  end
end

For the continuous priors introduced in the previous chapter, there are no matching criteria. This means that there is no need to find matching pairs of tuples in the memory. Instead, to find pairs of tuples in memory for computing the loss on the priors, one can simply sample a training batch, which is 2 times the training minibatch size. This batch can be split up to create the pairs of points, the same points can be used for each of the priors. Note that when sampling random observations from memory, the observations which are the last observation

Algorithm 5: Overview of the State Representation Learning framework

Initialise the state representation network ϕ with random weights
Define the number of episodes, E and the maximum number of steps per episode T
for *e in E* **do**
  for *t in T* **do**
    Predict state $\hat{s}_t = \phi(o_t)$
    Compute the action $a_t = clip(\pi(\hat{s}_t) + \mathcal{N}, a_{min}, a_{max})$
    Take action a_t in the environment and receive o_{t+1}, r_{t+1},and d_{t+1}
    Save an experience tuple to the SRL memory $R \leftarrow (o_t, a_t, r_{t+1}, d_{t+1})$
  end
  if *Update* ϕ **then**
    for *number of epochs* **do**
      for *Batches in epoch* **do**
        Create training batch according to matching algorithm
        Calculate the loss for each prior based on the sets of points in the batch
        Calculate regularization loss L_{reg} for all network weights according to L_2 regularization
        Compute $L_{\text{priors}} = \omega_1 L_{\text{temp}} + \omega_2 L_{\text{caus}} + \omega_3 L_{\text{prop}} + \omega_4 L_{\text{rep}} + \omega_5 L_{\text{reg}}$ Update parameters of ϕ using L_{priors}
      end
    end
  end
end

in the sequence are removed from the minibatch, as $\Delta\hat{s}_t$ cannot be computed, given there is no o_{t+1} in that sequence.

Neural network architecture

The state representation network encoder ϕ is implemented using a neural network. The state representation network computes the estimated state from the observation $\hat{s}_t = \phi(o_t)$. As mentioned previously, the robot uses two sensor modalities as observation, a camera and a lidar sensor. The observations of the camera are downsampled to 32x24x3, and the lidar sensor information is downsampled to 40 observations. To deal with these two different sensor modalities, the neural network is divided into two separate streams, one for the camera input and one for the lidar input. These streams are then later concatenated and used to compute the state estimate.

Figure 4.5 shows the architecture of the neural network used for the state representation. The convolutional layers (Conv) are defined as *channels* x *kernel size* x *stride*. Each of the convolutional layers in the network has a kernel size of 3 and a stride of 1. The number of channels are the number of output channels of that layer. The number in the Dense layers represents the number of nodes in the layer. Lastly, each block shows the activation function which is used. At the output of the network, gaussian noise is added, with $\sigma = 10^{-6}$. After each convolutional layer, a batch normalisation layer (23) is added, which helps to speed up and stabilise the learning process. The two streams are concatenated before going into the final Dense layer.

This network architecture is designed to work with the task of mobile robot navigation using the Gazebo simulator and the Turtlebot mobile robot. Other experiments in which only a camera observation require a different architecture. These experiments make use of a larger observation of 100x100x3. Therefore, pooling layers are introduced into the architecture to down sample the data.

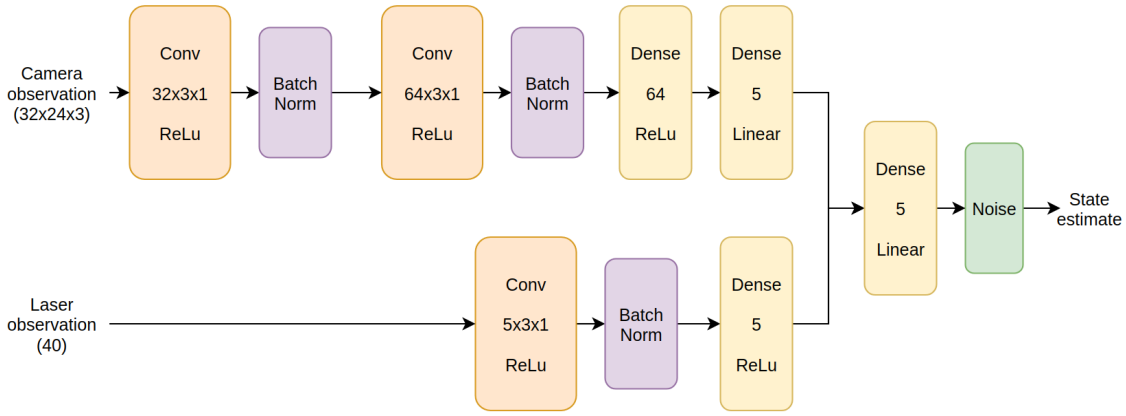


Figure 4.5: State representation network architecture overview

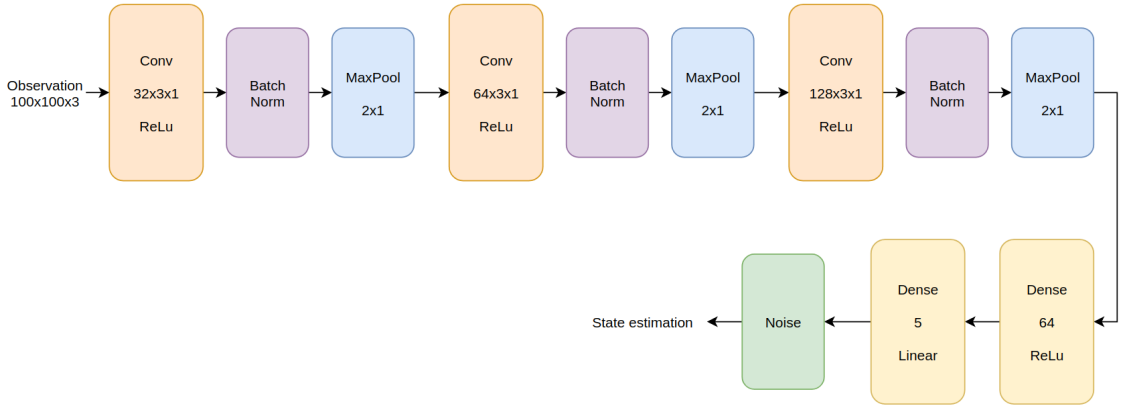


Figure 4.6: Architecture of the state encoder network used for non-gazebo environments with only rgb observations

Adaptation of the neural network architecture

The network architecture presented in Figure 4.5 is used in all experiments unless specified otherwise. This architecture fuses the two streams at the end of the network, with only a linear layer left to combine the two streams. This limits the way the network can combine the information from both streams. To make the network more expressive, an adaptation to this architecture is made.

The new architecture can be found in Figure 4.7. To make the architecture more expressive, the two streams are fused earlier. Instead of reducing both streams to the state dimension before fusion, the streams are fused with higher dimensions. After fusing the two streams, the network now has two non-linear layers, after which it is reduced to the state dimension in a linear layer. The number of filters for the lidar sensor data is increased to 32, allowing for extracting more information.

Hyperparameters

Here the parameters and the used values for these parameters will be displayed.

- Learning rate (α): Different learning rates are used for the discrete and continuous priors. For the discrete priors $\alpha = 0.001$ and for the continuous priors $\alpha = 0.0005$.
- Batch size: The batch size used for computing the loss on each of the priors was set to 256

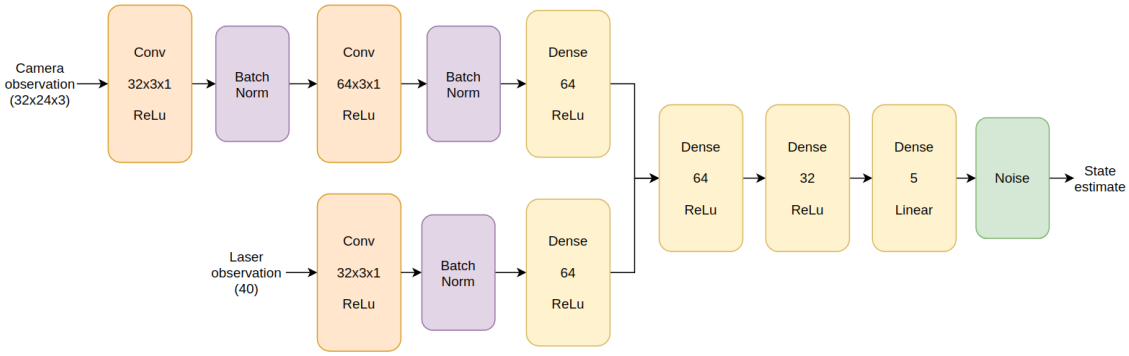


Figure 4.7: New architecture used for the state encoder network

- SRL memory size: The size of the SRL memory was set to 30000 experience tuples, although it was found that using a smaller memory size also gave good results.
- Epochs: The number of epochs for which the state representation network is trained was set to 20.
- Conditional parameters: These parameters are used in the conditional priors for finding pairs of observations for training.
  - ϵ: 0.01
  - κ: 0.01
  - μ: 0.1
- Continuous parameters: These parameters are used in the continuous priors.
  - α: 2
  - β: 10
- Prior weighting parameters ω, (conditional priors / continuous priors):
  - Temporal coherence prior weight (ω_1): 1/1
  - Causality prior weight (ω_2): 5/1
  - Proportionality prior weight (ω_3): 5/1
  - Repeatability prior weight (ω_4): 5/1

4.2.3 Adapting the state representation framework to work with recurrent neural networks

The previously laid out details on the implementation of the state representation framework have not been designed to deal with recurrent networks. To answer research question 2, the framework needs to be adapted to be able to work with recurrent neural networks. This entails changing the neural network itself to add recurrency, as well as adapting the structure of the state representation memory and the way in which training batches are sampled.

For the training of recurrent neural networks (RNNs), sequences of samples are used instead of just a collection of samples. In the non-recurrent setting, a minibatch of samples is sampled from the memory for training the neural network, as this reduces variance and increases training stability. Also, the batch normalization layers in the neural network need a sufficiently large batch size to give a good normalization. A minibatch is a collection of samples from the memory, which has been collected randomly, distributed uniformly over the entire memory.

When training a RNN, a minibatch is not a randomly sampled collection of single samples, instead the minibatch will consist of a randomly sampled collection of sequences. The sequences of experiences which are stored in the memory are not all of the same length. Training the model using a batch of sequences requires some adaptations to the way these experiences are stored.

Adapting the memory

When it is not important to keep track of the sequences in the memory, the experiences can easily be added to a simple array. This is simple in implementation and relatively efficient. When it is important to keep track of the sequences in the memory, the architecture becomes a bit more complex. One could save sequences of experiences into the memory directly, however as these sequences all have varying length, this does not allow the use of standard numpy arrays. Instead, one could pad each sequence to have the length of the maximum number of steps per episode. The downside of this would be the additional memory this requires. Although probably not a problem on modern machines when dealing with medium sized memory sizes, this could become problematic when using large memory sizes or large potential sequences.

Instead, the experience tuples are still saved into a single array. To keep track of the start and end of each sequence, a separate array is used to keep track of the length of each sequence. This array contains a value for each of the sequences of the array, indicating the length of the sequence. This way, the arrays containing the experience tuples can be predefined, reducing the memory requirement and speeding up the code. The start and end points of each sequence can be found using the sequence lengths.

If the maximum memory size has been exceeded, the first sequence in the memory is be purged. This way the most recent sequences of experiences are in the memory.

Sampling and padding

Training the neural network using a batch of sequences requires these sequences to all be of the same length. However, these sequences have varying lengths, as episodes have different lengths. Furthermore, the sequences can be 150 steps long. Using such sequences has a significant effect on the speed of training. Therefore, the sequences will be truncated to the same length for each batch. The length at which the sequences will be truncated is a training parameter which can be varied to trade of the speed of trading and the effective memory length.

A common way to ensure equal length sequences in the batch is to truncate the sequences if possible, and otherwise pad the sequence with zeros. This means padding the beginning of each sequence with zeros if the point is in the beginning of the sequence. However, a problem with padding these points with zeros is that due to the biases in the network, the internal state of the memory will not be zero at the first experience in the sequence.

Instead of padding the sequences, the points from the beginning of each sequence are not used for training. Instead, these points will only be used for training as part of the sequence of another training sample.

Change the architecture of the NN to include LSTM

The architecture used in the non-recurrent state representation learning is adapted to be recurrent by including LSTM layers (46). In line with the work done by (36), two LSTM layers are added. The number of units in the hidden state was set to 256. Dropout layers were added after the LSTM units to prevent overfitting.

Hyperparameters

Again, several parameters are involved in training the state representation. These parameters are largely the same parameters which were introduced for the non-recurrent state represen-

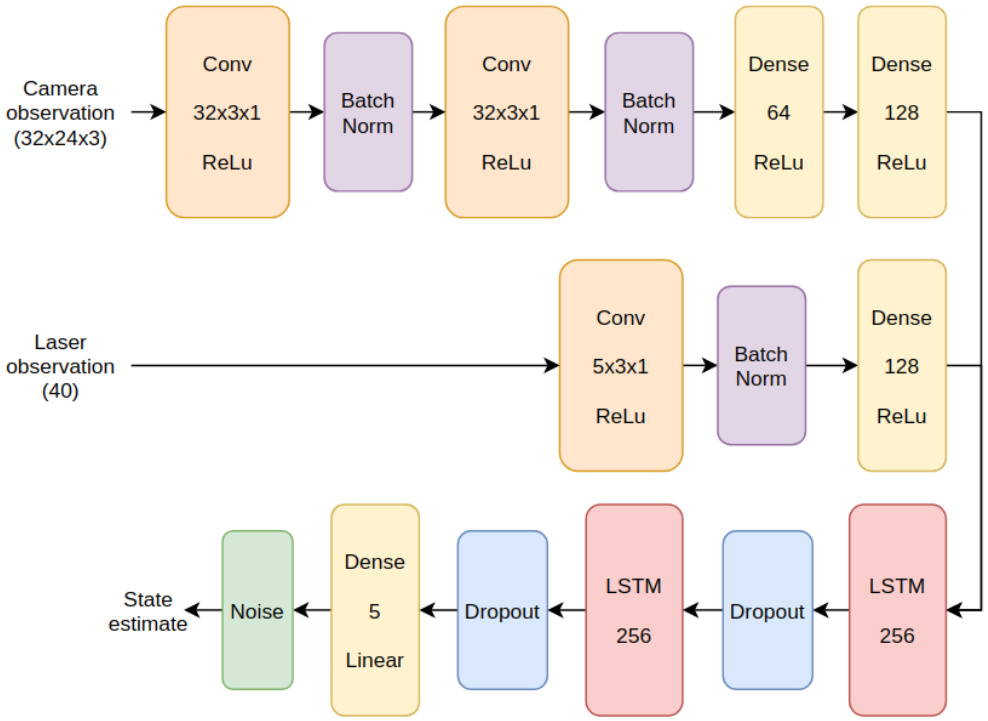


Figure 4.8: State representation network with recurrent architecture overview

tation learning framework. These parameters had to be re-tuned to work well with this new architecture.

- Sequence length: The sequence length is changed between environments and is tuned for the best results.
- Learning rate: 0.0003
- Batch size: 128, when the sequence length is longer than 16, the batch size and learning rate are decreased by a factor of 2 for each doubling of the sequence length.
- SRL memory size: 30000
- Epochs: 30
- Prior parameters: These parameters are used in the continuous priors.
  - α: 2
  - β: 10
- Prior weighting:
  - ω_1: 0
  - ω_2: 1
  - ω_3: 1
  - ω_4: 1

4.3 Simulation setup

This section will cover a more in depth look into the simulation setup which is used for this work. The simulator which is used is the Gazebo simulator. Gazebo is an open-source simulator for robotics. The Gazebo simulator is well known and has models of some common robots, like the Turtlebot robots. The Gazebo simulator further offers support for several sensor modalities, which includes the support for cameras and lidar sensors. The Robotic Operating System (ROS) (5) is integrated into Gazebo, allowing for communication between python and the Gazebo simulator. ROS works as the middleware, enabling communication between the two platforms. A openAI gym wrapper is then used to structure the interaction between the environment and the python code.

The state representation learning method and reinforcement learning algorithm are implemented in python. The reinforcement learning algorithm will generate actions using the learned policy, which are composed of a targeted translational and rotational velocity. This can be seen as high-level commands to the robot. Through the gym interface, and the ROS middleware, this command is sent to the low level controller of the Turtlebot robot. Apart from controlling the wheels of the robot. The observations are read on the robot, and sent to the gym interface, using the ROS middleware.

The low-level controller and the reading of the sensors of the robot is done in the Gazebo simulator. This is inspired by a realistic setup, as in a real setup, this would be done on the robot.

Two different sensors are used for learning the state representation, which are a camera and a lidar sensor. The camera on the Turtlebot3 robot which is used has a 680x480 pixel RGB camera. Given that the environments are created using coloured walls without any texture, this high a resolution is not necessary. Using a much smaller image from the camera can give a loss of detail, but will make computation much faster, which is a significant advantage. The images from the cameras are downsampled to 32x24 RBG images, which is sufficient for this task. The camera has a field of view of 62.2 degrees in the horizontal direction.

A second sensor modality which is used is the lidar. This sensor has 360 laser distance measurements, which is distributed over 6.28 radians. The maximum range this sensor can measure is 3.5 meters, which is equal to the real world laser sensor which is on the Turtlebot3 waffle pi. As it is not necessary to have so many samples, this is downsampled to 40 measurements, evenly divided over the 6.28 radians.

The lidar sensor is used to check for collisions, and the robot is considered to have collided with one of the walls, if the sensor measures less than 0.26 meters between the sensor and the wall.

4.4 Summary

This chapter has first covered the experimental design of this work. This section has covered the design of the experiments and has introduced the environments in which these experiments have been performed. Then a detailed look into the implementation of the algorithms for both the RL algorithm as well as the state representation learning algorithm is given. It has covered the different implementations of the robotic priors to facilitate learning the state representation using the different priors. Finally, the simulation setup was covered.

5 Results and Discussion

This chapter will cover the results from the performed experiments, which are used for answering the research questions. All results will be covered and interpreted. First, some ground truth baselines will be presented. Then the results for answering research question 1 will be covered in detail. These results will be split up between the two parts of the first research question. The first part will cover the priors in set one and two. The second part will consider the continuous priors in sets three and four. These different sets of priors have all been tested on the environments as discussed in the previous chapter. The performance and the resulting state representations will be compared and discussed. Then the results for research question two will be presented.

5.1 Ground truth baseline

The ground truth baselines will serve to show what performance should be expected from the training a policy if the state representation is very good.

5.1.1 Training settings

The ground truth state has a dimensionality of five, which is equal to the dimensionality of the learned state representation. This is important for the comparison, as the dimensionality of the input state can be important for the performance of the RL algorithm, where a high input dimensionality typically makes solving a task harder.

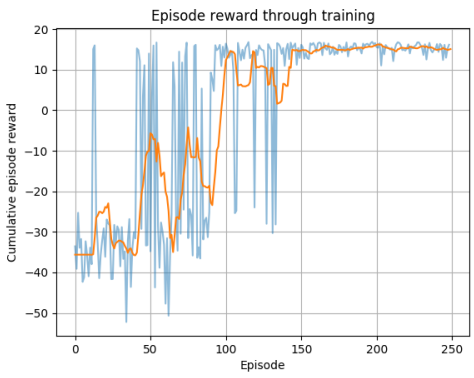
The ground truth state vector is composed of the following information: the x coordinate, the y coordinate, the orientation of the robot in radians, the x position of the target, and the y position of the target. The input vector thus becomes $\left(x_{\mathrm{r}}, y_{\mathrm{r}}, \theta_{\mathrm{r}}, x_{\mathrm{t}}, y_{\mathrm{t}}\right)$. The coordinates of the target position is not strictly necessary, given that the target location does not move. The agent can therefore simply learn the policy to move to the target based on the reward signal, and does not need the location of the target in its state. However, keeping this information in the state gives that the dimensionality of the input state is equal between using the ground truth data and the state estimate.

The exploration noise is kept equal to the exploration noise in the other experiments, where the learned state representation is used to learn a policy. The actor and critic networks are initialised randomly. The agent is trained for 250 episodes for each environment, the magnitude of the noise is linearly decreased between the first and last episodes.

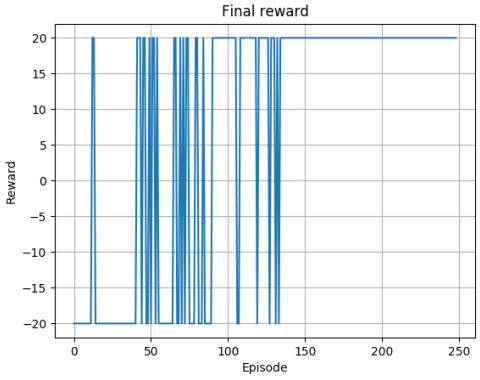
5.1.2 Results small square environment

When trained on the ground truth state in the small environment, the agent is able to converge to a good policy within 150 episodes. Possibly this could be even quicker if the hyperparameters of the DDPG algorithm would be tuned further.

Figures 5.1a and 5.1b show the results in this environment. The blue signal in Figure 5.1a represents the unfiltered episode reward, the orange line is the moving average across 10 episodes. The goal of the agent is to optimise its cumulative reward. Figure 5.1b shows the final reward received by the agent, the reward at the terminal step. This reward is -20 if the agent has hit the wall, close to zero if it has reached the maximum step and +20 if the agent has reached the target. In the large square environment this is set to -50 and +50. From these two graphs it can be seen that the final reward dominates the cumulative reward per episode, this is especially true since the environment requires only few steps to reach a solution. From Figure 5.1b with the final reward, one can clearly see that from episode 150 onwards, the robot reaches the target each episode and is always successful. In these episodes, there is still exploration noise added,



(a) Episode reward throughout the training process



(b) The final reward received per episode

.

Figure 5.1: Training results using the ground truth state in the small square environment

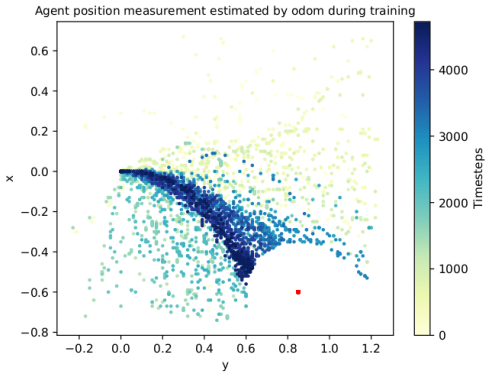


Figure 5.2: Points visited by the robot throughout training using the ground truth in the small square environment

which can be seen in graph of cumulative reward. One can see that the cumulative reward still varies, which indicates that the agent does not take the same path in each episode. Despite the noise, the agent is still able to reach the target each episode.

Figure 5.2 shows the (x_{r_t}, y_{r_t}) coordinates visited by the robot throughout training. The darker the points, the later these have been visited in the training process. One can see that even on the darkest paths, there is still some variance in the taken path to the target location.

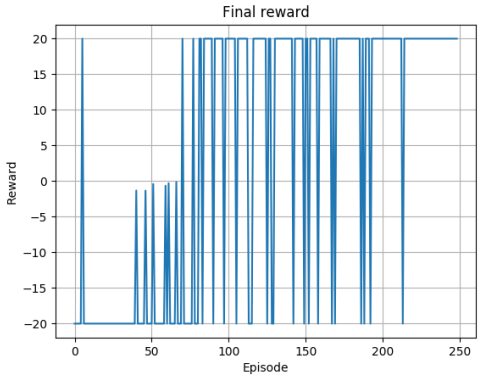
Unlike some of the other reinforcement learning algorithms, DDPG is deterministic. If no noise is added, and the agent would not be trained further, it would perform the same actions each episode.

5.1.3 Results L-shape environment

The L-shape environment is slightly more complex, as the agent is more restrained in which path to the target can be taken, as well as the fact that this environment is larger. This means that more steps are needed to solve the task, which typically makes training harder.

Given that more steps are needed to reach the target, also means that the agent is more affected by the accumulating noise. This is especially true since the noise which is added in an episode does not have a zero mean, and is thus working against the policy over more steps. From Figure 5.3b, which shows the terminal, or final reward, can be seen that the agent learns to reach the target relatively quickly, but fails to reach the target in some of the episodes. As the magnitude of the noise decreases, the agent learns to reach the target more consistently, where after episode 210, the agent reaches the target each episode. The same effect can be seen in Figure

(a) Episode reward throughout the training process



(b) The final reward received per episode

.

Figure 5.3: Training results using ground truth in the L-shaped environment

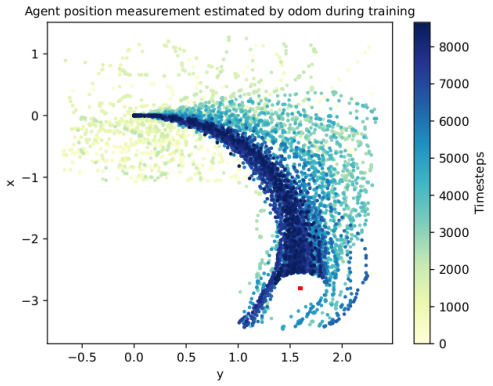


Figure 5.4: Points visited by the robot throughout training using ground truth for the L-shaped environment

5.3a, which shows the cumulative loss per episode. The points visited by the robot for the L-shaped environment can be found in Figure 5.4. From this figure, it can be seen that the agent misses the target location on some occasions.

5.1.4 Results large square environment

The large square environment requires many steps to successfully complete. The reward function was therefore adapted to have a reward of -50 and 50 for hitting the walls and reaching the target respectively. The biggest challenge of this environment is the size, the policy will need to learn its way through the entire environment.

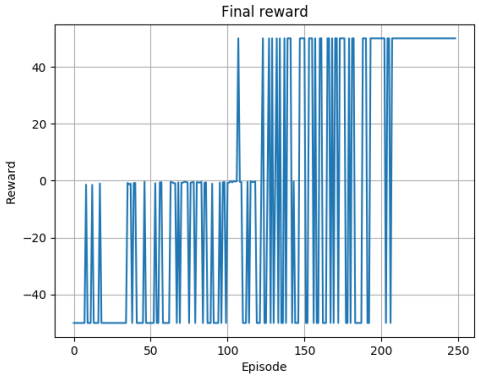
Figure 5.5a shows the cumulative reward per episode. Figure 5.5b shows the terminal reward for each episode. unlike the small and L-shaped environments, the large environment shows relatively many terminal rewards close to zero. This means that there are relatively many episodes in which the agent does not reach the target but also does not hit a wall. This is as expected, given that the environment is a large open space. In the last episodes the agent does learn to reach the target reliably.

5.2 RQ1

5.2.1 Testing the adapted priors from literature

As discussed in chapter 3, the priors proposed by (24) and (11) have been adapted to work with a continuous action space. These priors have both been implemented. These priors have some hyper-parameters, which have been tuned to work well across the different environments. Tun-

(a) Episode reward large square environment



(b) The final reward received per episode for the large square environment

Figure 5.5: Training results using the ground truth state in the large square environment

ing these parameters for each of the environments could potentially improve the learned representation, however in my experience, this did not make a significant difference to the results. Furthermore, tuning these parameters is a very laborious job, which would make jumping to a new environment a non-trivial task.

For this reason and the reason that the priors should be easy to use between different environment, a set of hyper-parameters is found that gives a good compromise between the performance for the environments. These hyper-parameters are then kept constant between the different environments.

For each of the environments a dataset containing at least 30000 samples is gathered. The memory size of the state representation framework is kept at 30000, and as such, an equal number of samples will be used for each of the environments. The state representations resulting from different sets of priors are trained using the same training data.

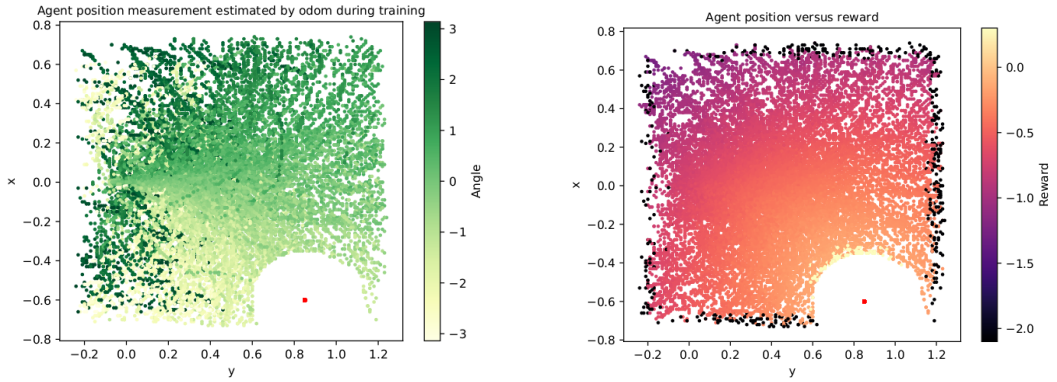
Prior set one

This will cover the results which have been obtained with the action based priors, which are adapted from (24). These priors are defined in equations 3.1, 3.8, 3.9, and 3.10.

Small environment

First the small square environment will be used to test these priors. To train the state representation, the dataset for the small environment is used. The distribution of observations over the environment can be seen below in Figure 5.6a. As can be seen from this figure, the data is well spread out over the environment. Figure 5.6b shows the same (x_r, y_r) points as figure 5.6a, but then shows the reward the agent received at that point by the color of the point. The reward is clipped at -2.0 and 0.1 in the plot, such that the rewards for reaching the target and hitting the wall is distinct but does not dominate the range. As can be seen clearly, the reward function changes smoothly over the environment, this smoothness should be present in the learned state representations as well.

To quantitatively judge the learned state representations, two methods for plotting this data will be mainly used. These are a PCA plot of the first two principal components, and a t-SNE plot, which brings down the dimensionality to 2, such that it can be plotted. The reason for using both, is that these plots can give different insights, and will complement each other. Using PCA to plot the first two principal components has the advantage of it often retaining some part of the shape of the environment. This method however has the disadvantage that it can map points that are not actually close in the high dimensional space close to each other. This will be especially true if the data has many principal components. The t-SNE plotting method has the



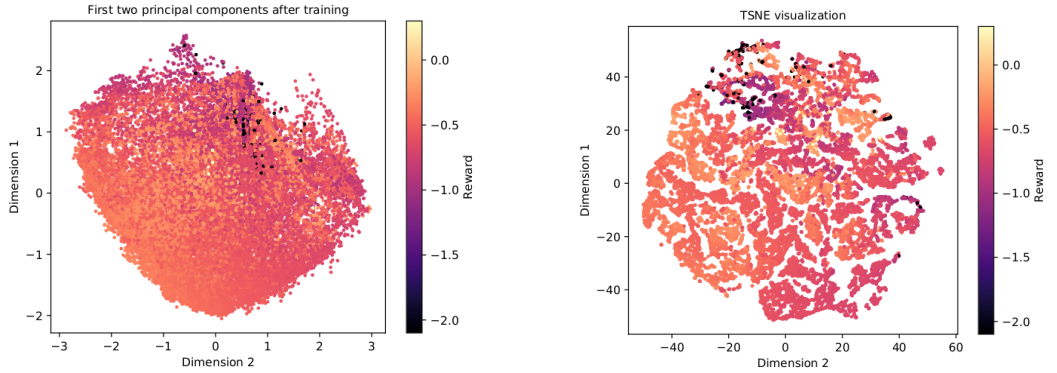
(a) The points of observations in the dataset of the small square environment

(b) The reward values for the points of observations in the dataset

Figure 5.6: The training dataset for the small square environment

downside of completely losing the physical structure of the environment. The advantage of this plotting method is that it will better retain the relation between points in the high dimensional space, mapping close points close together.

The plots of the first two principal components and the t-SNE plot of the learned representation of the small square environment can be found in Figures 5.7a and 5.7b respectively. From the plots can be seen that there is no clear color gradient in either plot. This indicates that the learned state representation is not smooth, making learning a good policy more difficult.



(a) First two principal components of the SR for the small environment, prior set one

(b) T-SNE visualization of the SR for the small environment, prior set one

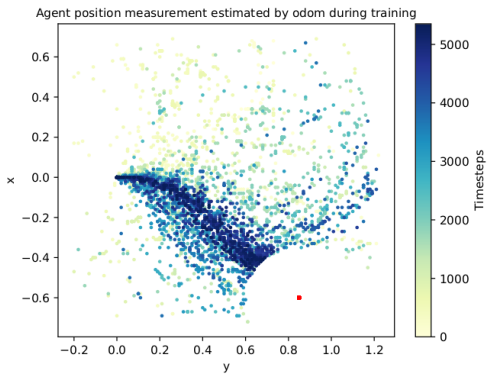
Figure 5.7: Learned representation using prior set one on the small square environment

The goal of the learned state representation is to allow the RL policy to efficiently learn a good policy. Therefore, the learned state representation should be judged by the ability of the RL algorithm to learn a good policy. Figure 5.8b shows the moving average of the cumulative reward per episode. Figure 5.8a shows the points visited by the robot during training. From the reward plot and the path of the robot can be seen that a good policy is found and the target is reached reliably.

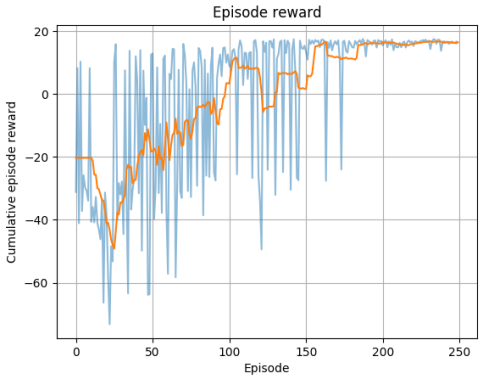
L-shaped and large environments

As discussed, this set of priors worked well on the small, and easiest environment. These priors are also tested on the L-shape environment and the large square environment. The reward per episode for these environments can be found in figure 5.9a and 5.9b respectively.

From the episode reward in the L-shape environment in Figure 5.9a can be seen that a good policy is found, but then lost when the magnitude of the noise tapers off. From Figure 5.9b

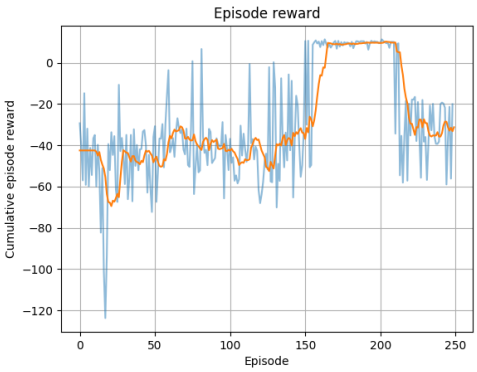


(a) Visited states during training



(b) Episode reward during training in the small environment

Figure 5.8: Training results using prior set one in the small environment



(a) Cumulative reward during training on the L-shape environment

(b) Episode reward during training on the large square environment

Figure 5.9: Training results using prior set one in the L-shaped and large environments

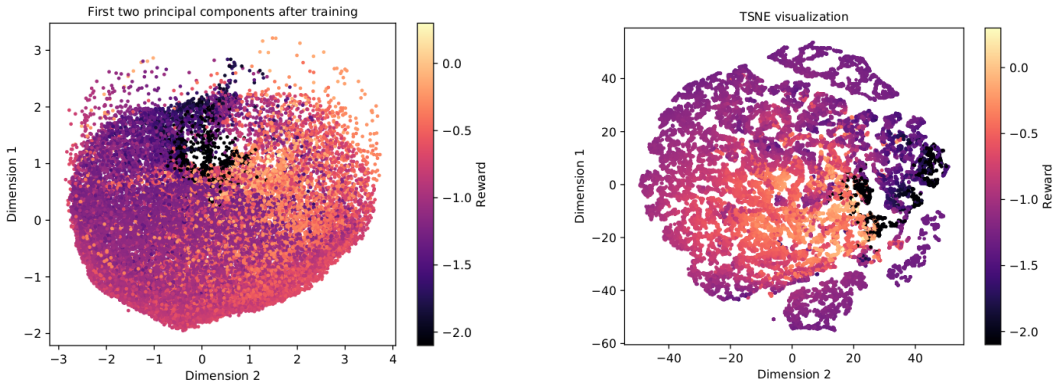
showing the cumulative reward of the agent in the large square environment can be seen that at no point during training the agent is able to reach a good policy. The learned state representation for the large environment is visualised in Figure 5.10. From these visualisations it can be seen that the points close to the target and points far away are very close in the learned representation. A similar effect can be seen in the L-shaped environment shown, which can be found in Appendix B. This shows that states that are far apart are not pushed apart sufficiently using this set of priors.

Prior set two

The second set of priors introduced in chapter 3.2.1, are the priors which replaced the condition of a similar action with the condition of similar reward variation for the proportionality and repeatability prior. The priors are defined in equations 3.1, 3.9, 3.5, and 3.6, for the temporal coherence prior, the causality prior, proportionality prior and repeatabilty prior respectively.

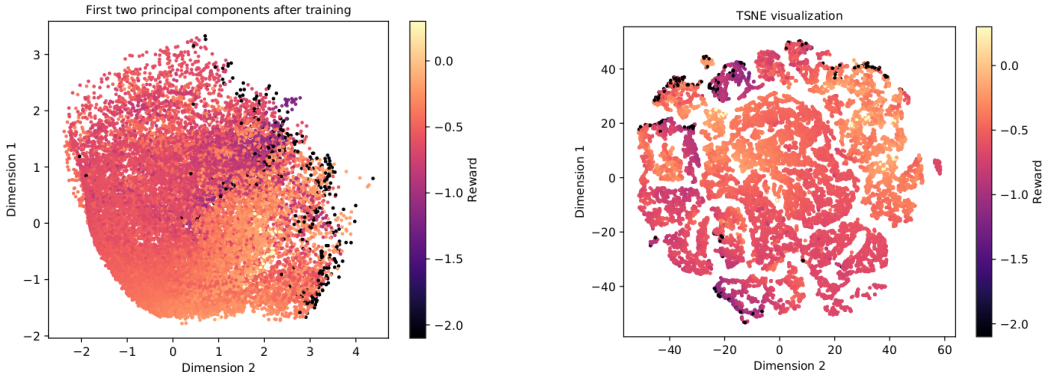
Small square environment

Comparing the learned state representation using the reward variation based priors and the action based priors shows a clear difference. The t-SNE plot in Figure 5.11b shows a much clearer color gradient, and a much smoother representation than the plot in Figure 5.7b. Also the walls, which are the black dots, are grouped better, both in the t-SNE plot, as well as in the first two principal components shown in Figure 5.11a.



(a) First two principal components of learned SR **(b)** T-SNE visualisation of the learned SR

Figure 5.10: Visualisation of the learned SR using prior set one in the large square environment



(a) First two principal components of the SR of the small square environment, prior set two **(b)** T-SNE visualisation of the SR of the small square environment, prior set two

Figure 5.11: Learned representation of the small environment using prior set two

From Figure 5.12b which shows the moving average of the cumulative reward per episode during the training process, we can see that using these priors, a good policy can be learned. Comparing this reward signal with the cumulative reward when training on the ground truth data, as displayed in Figure 5.1a, it can be seen that the speed of convergence is very similar. Also, the reward per episode is very similar after it has converged.

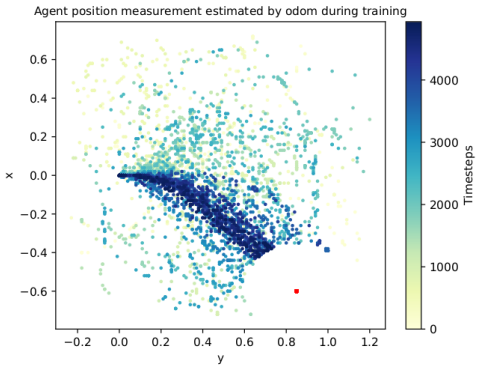
L-shaped and large environments

Figure 5.13 shows the episode rewards for both the L-shaped and large environment. In the L-shaped environment, a good policy can be learned, although the policy does not learn to take the shortest path. The path the robot takes can be found in appendix B, as well as a visualisation of the learned state representation. For the large environment, no good policy can be found. When looking at the first two principal components and the t-SNE visualisation of the learned SR in the large environment, found in Figure 5.14, it can be seen that the learned representation is not very smooth. Looking at the t-SNE representation it can be seen that there is no clear color gradient, and there is some mixing of colors. This points to a bad representation as the cause of not learning a good policy.

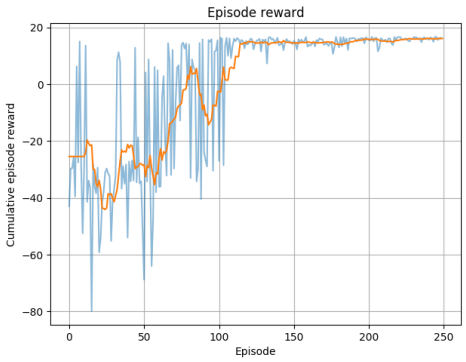
5.2.2 Testing the continuous priors

Initial results

The initial results using the priors of sets three and four did not show promising results. The representation tended to have all information in only two principal components in the large

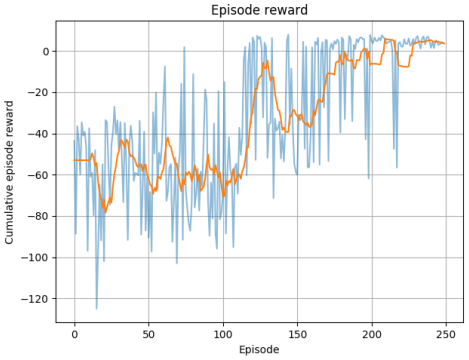


(a) Robot locations during training in the small square environment, prior set two

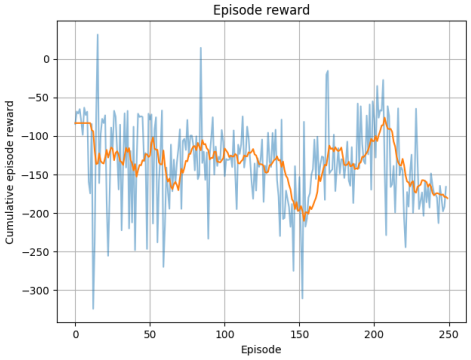


(b) Cumulative reward during training in the small square environment, prior set two

Figure 5.12: Training results in the small environment using prior set two

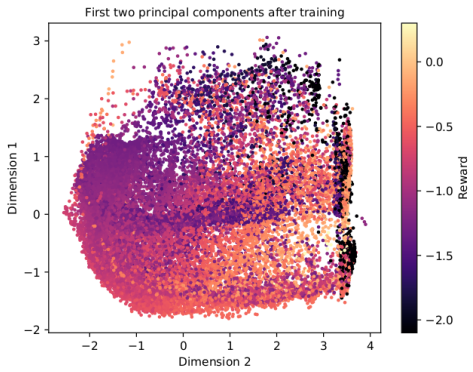


(a) Episode reward during training using prior set two in the L-shaped environment

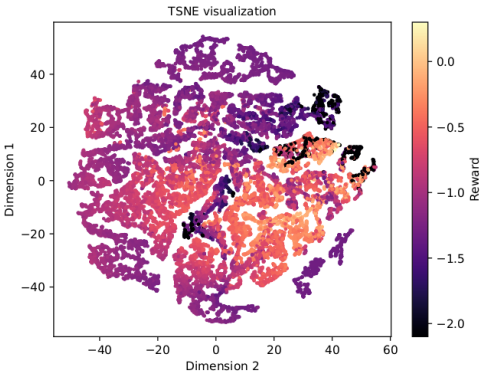


(b) Episode reward during training using prior set two in the large square environment

Figure 5.13: Training results in L-shaped and large environment using prior set two



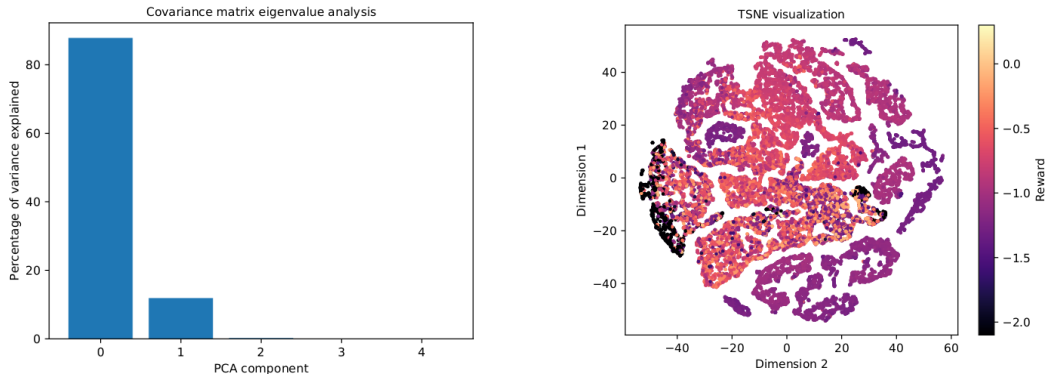
(a) First two principal components of the large environment using prior set two



(b) T-SNE visualisation of the large environment using prior set two

Figure 5.14: Visualisation of the learned SR in the large environment using prior set two

square environment. The resulting state representation was not smooth, nor well separated. The variance explained per principal component of the learned representation can be found in Figure 5.15a. This graph shows that most variance of the learned SR is expressed only in a single dimension. The t-SNE visualisation shown in Figure 5.15b of the learned representation shows that many states are not separated properly.



(a) Variance explained by principal component for the learned SR

(b) T-SNE visualisation of the learned SR in the large square environment using prior set three

Figure 5.15: Initial results using prior set three in the large environment

This was worst in the large square environment, because the reward difference between different states is larger on average than the other environments due to the larger size. This could be countered by decreasing a in the reward function shown in 4.3. However, decreasing a can have a detrimental effect on the performance of the DDPG algorithm. Having large differences in the rewards gives that the term $|r_{t+1} - r_{t+2}|$ dominates the loss, as this term is not bounded while the exponential terms of the loss function are bounded between 0 and 1. Because of this, the term $|r_{t+1} - r_{t+2}|$ was dropped from the causality prior loss function. The new causality prior is defined in equation 5.1.

$$L_{\text{caus}}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2} e^{-\beta\|a_{t_1}-a_{t_2}\|^2}\right] \tag{5.1}$$

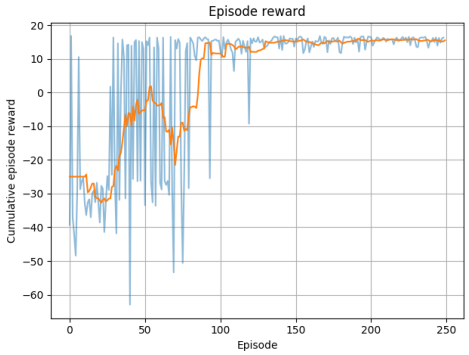
Prior set three

The action based continuous priors are defined in Chapter 3. The temporal coherence prior, proportionality prior, repeatability prior, and causality priors can be found in equations 3.11, 3.12, 3.13, and 5.1 respectively.

This set of priors is able to perform well in each of the three environments. The cumulative reward achieved during the training process can be found in Figure 5.16. Figure 5.16a shows the episodic reward achieved in the small square environment. This reward signal shows a similar pattern to training the policy on the ground truth data. A good policy is found in 100 episodes, and then converges. The L-shaped environment is more challenging, the reward signal can be found in Figure 5.16b. This reward signal shows that a good policy is found early on in training, but then lost again. However, it does recover again at the end of the training period. When comparing this to the policy trained on the ground truth state, of which the reward signal is displayed in figure 5.3a, it can be seen that the reward signal is more stable when training on the ground truth data. The policy trained on the large environment shows relatively quick convergence to a good solution. The t-SNE visualisations of the learned state representations for each of the environments can be found in Appendix B.

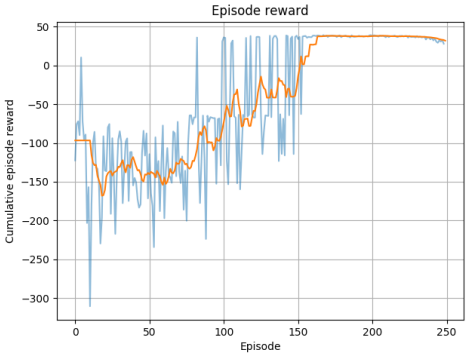
Prior set four

Replacing the term $e^{-\beta\|a_{t_1}-a_{t_2}\|^2}$ in the proportionality prior and repeatability prior with the term $e^{-\beta\|\Delta r_{t_1}-\Delta r_{t_2}\|^2}$, showed similar results. Where the performance of the conditioned priors showed significant difference between using the reward based conditioning versus the action based conditioning, this did not translate to the continuous priors. The action based conditioned priors from prior set one did not show a strong performance, given that it did not allow for solving two of the three environments. Figure 5.17 shows the reward for each of the three



(a) Episodic reward for the small square environment

(b) Episodic reward for the L-shaped environment



(c) Episodic reward for the large square environment

Figure 5.16: Episode reward in three environments using prior set three

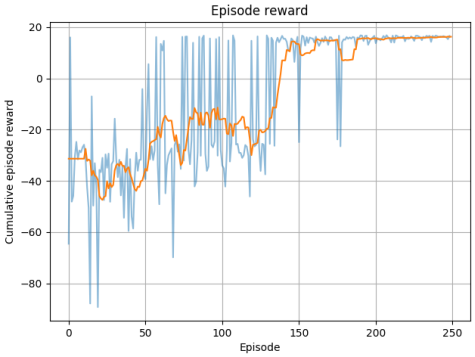
environments. The t-SNE visualisations of the learned state representations for each of the environments can be found in Appendix B.

5.2.3 Comparing the performance of the priors

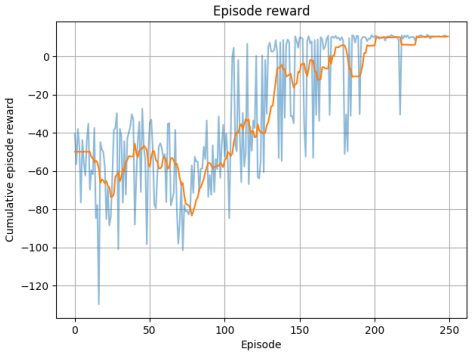
Now that the different sets of priors have been tested, the performance of the priors can be compared. It was found that adapting the originally proposed priors with the condition based on actions, prior set one, gave poor performance on the L-shaped and large square environments. Changing the condition on the repeatability and proportionality priors to a condition based on the reward difference significantly improved the results, as can be seen in section 5.2.1. This second set of priors showed good performance on the small square environment and the L-shaped environment. On the large environment, the performance deteriorated.

Given that these priors did work well on the other environment, these priors could likely be made to work well in the large environment as well. This would however require re-tuning the hyper-parameters to make these priors work in the large environment. Tuning hyper-parameters can be a costly endeavour, as it requires multiple training runs for both the state representation as well as the RL algorithm.

Then the continuous priors were tested, which were designed specifically with a continuous action space in mind. These priors removed the requirement of finding observation pairs that adhered to the condition on the priors, simplifying the implementation. Comparing the continuous priors based on the action condition in section 5.2.2 and the priors partly based on the rewards in section 5.2.2 does not show any major differences. Both sets of priors allow the agent to learn a good policy. Differences in the speed of learning between the two sets of priors can be attributed to differences between runs, as similar differences were observed between different runs with the same priors.



(a) Episodic reward for the small square environment



(b) Episodic reward for the L-shaped environment

(c) Episodic reward for the large square environment

Figure 5.17: Episode reward in three environments using prior set four

5.3 RQ2

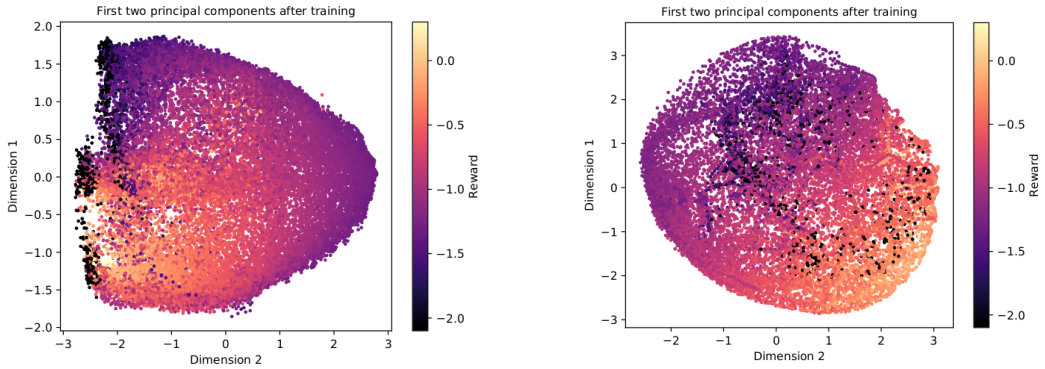
This section will cover the experiments that have been performed to answer research question 2. The recurrent state representation learning framework is tested on multiple environments, as can be read in section 4.1. The results from these experiments will be presented in this section, comparing the performance of a recurrent framework versus the standard framework. At first, the large square environment will be covered. Then, the results from the other environments will be presented. All experiments use the robotic priors in prior set three, as used in section 5.2.2.

5.3.1 Large square environment

The results for the large square environment have been presented in the previous section. This section has covered only the results for one target location one, positioned close to a corner. The fact that it is close to a corner is significant, as this makes the locations around the target easier to locate. To test the effectiveness of the LSTM framework, a second goal is added. The locations of both goals can be found in Figure 4.2. This second goal is placed close to one of the walls, far removed from any corners. This makes locations around the target more difficult to determine, as the observations from the camera are not ambiguous.

Target location one

In the previous section, it was shown that the non-recurrent state representation framework was able to learn a state representation that enables learning good policy. This section will further expand on these results and introduce the results from learning the state representation using the recurrent state representation learning framework.

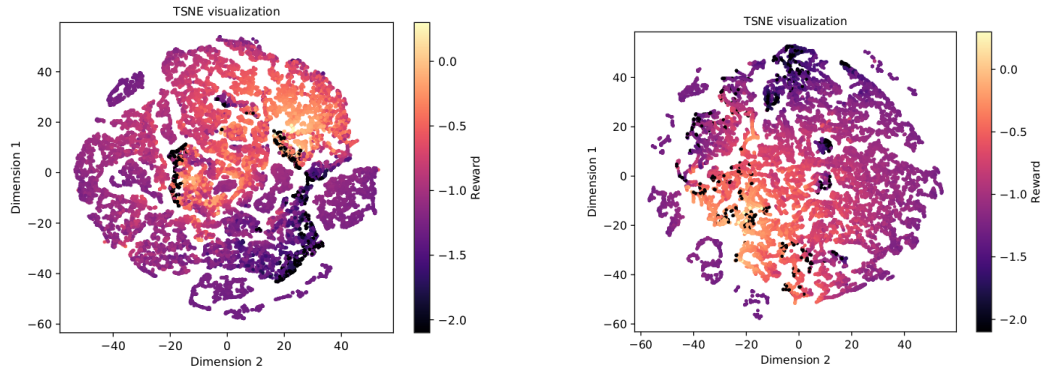


(a) First principal components of the learned SR using the non-recurrent state encoder

(b) First principal components of the learned SR using the recurrent state encoder

Figure 5.18: First two principal components of the learned representation on the large environment

The first two principal components of the learned state representation when using a non-recurrent network to learn the state representation are shown in Figure 5.18a. In this representation it can be seen that the black points, which represent the locations of the walls, are grouped together in clusters. Three different clusters can be distinguished, representing the three walls of the environment which are hit most often. The first two principal components of the learned state representation using the recurrent network can be found in Figure 5.18b. Comparing this to the non-recurrent state representation it can be seen that this representation shows a much smoother transition in the colors.



(a) T-SNE visualisation of the learned SR using the non-recurrent state encoder

(b) T-SNE visualisation of the learned SR using the recurrent state encoder

Figure 5.19: T-SNE visualisations of the learned representations of the large environment

The t-SNE representations for both the non-recurrent and recurrent state representations can be seen in Figures 5.19a and 5.19b respectively. In the plot showing the visualisation of the non-recurrent state representation, it can be seen that the different areas of the environment are separated well, as it shows good separation of the colors. Comparing this to the recurrent state representation however, it can be seen that the non-recurrent state representation lacks the smooth color gradient that the recurrent state representation shows.

From the plots showing the non-recurrent state representation it can be seen that especially the states close to the walls of the environment are not represented as well as in the recurrent state representation. Therefore, it is interesting to see what the learned state representation looks like at observations close to the walls. To that end, the first two principal components from the state representation from the points seen in Figure 5.20 are mapped separately. These

mappings can be seen in Figure 5.21a and 5.21b for the normal state representation and the recurrent state representation respectively.

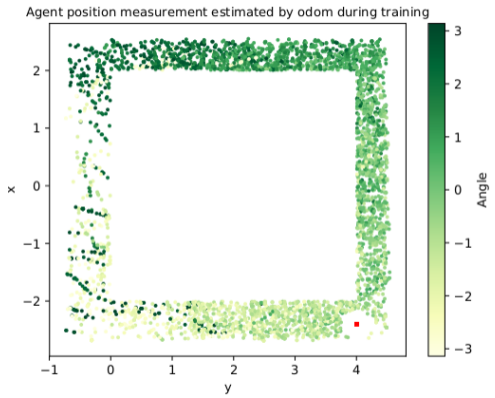


Figure 5.20: Robot positions close to the walls

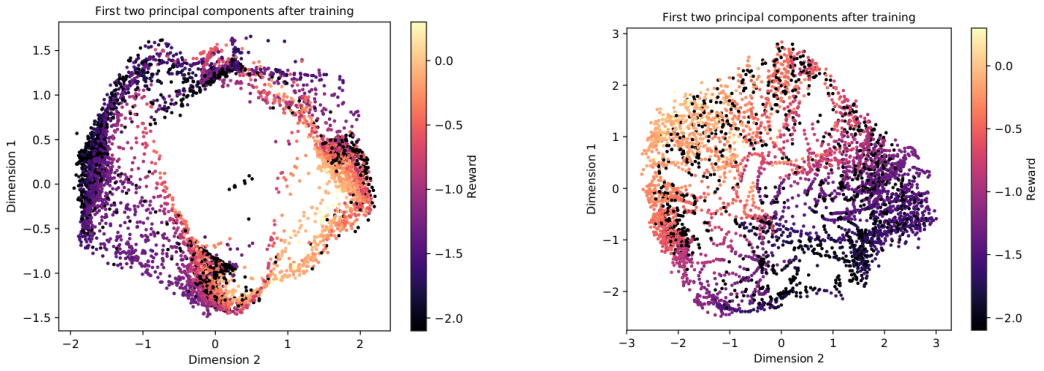
Looking at the state representation near the walls for the non recurrent state representation framework, as can be seen in Figure 5.21a, it can be clearly seen that it has four clusters of points. These clusters are the same as which can be seen in Figure 5.18a, which shows the state representation for the entire dataset. It can be seen in the figure that these clusters are interconnected with some points, but do not show a clear gradient in the color of these points. These interconnecting points are the observations in which the intersection of two walls can be seen in the camera observation. The target location, which are the brightest points, is also partially mapped between two clusters. The four clusters each represents one of the walls, and the points in between the corners between these walls. The representation shows an empty area in the middle, which indicates that it is able to retain some of the physical shape, as these points are all close to the walls.

Mapping the same points in the dataset using the recurrent state representation network, as can be seen in Figure 5.21b, shows a very different image. There are no real clusters of points, instead, the points are much more spread out. It can also be seen that the color of the points changes smoothly over the outer perimeter. This shows that the recurrent model is able distinguish the states well, in contrast with the non recurrent model. The states are not very well confined to the edges, as they are sometimes spread to the middle. This suggests that the recurrent state representation is not as good in keeping the physical properties of the environment.

Figure 5.22a shows the episode reward in the large square environment for target location 1. This plot shows the episode reward for both the non-recurrent and the recurrent state representation mappings. Both the recurrent and non-recurrent state representations allow for the policy to learn to reach the target. The non-recurrent state representation gives a more reliable policy, which converges to a good solution. The policy based on the recurrent state representation also learns to reach the target, albeit less reliably. In the last 100 episodes, the policy based on the non-recurrent state representation reaches the target 98 times, while the recurrent state representation allows the policy to reach the target 72 times in the same episodes.

Figure 5.22b shows the episode reward in the large square environment for target location 2. This target proves to be harder to reach, with the target reached only 56 and 50 times for the non-recurrent and recurrent based policies respectively. Furthermore, policies trained on a non-recurrent state representation proved to be unstable, with the RL algorithm failing to find a policy that reaches the target regularly. Across three attempts, the results from the policy shown in Figure 5.22b was the only policy that learned to reach the target regularly.

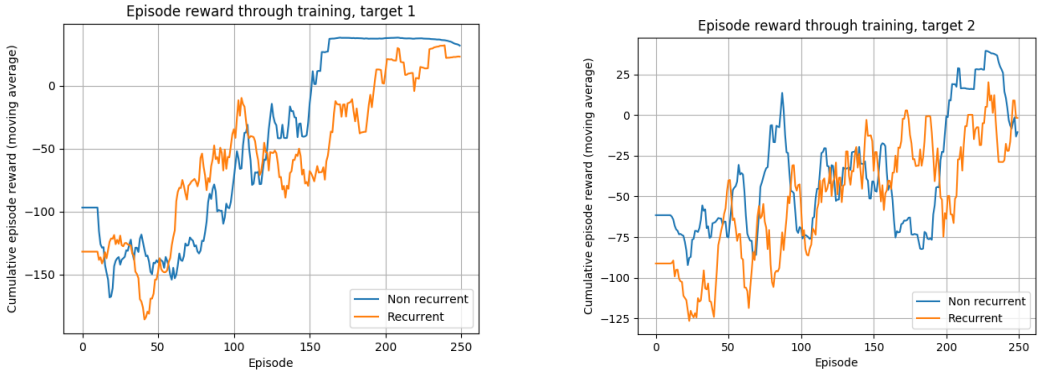
To analyse why the policy based on the recurrent state encoder network performs worse than the policy based on the non-recurrent state encoder for reaching target location one, further



(a) First two principal components of the non-recurrent state representation of the points from Figure 5.20

(b) First two principal components of the recurrent state representation of the points from Figure 5.20

Figure 5.21: First two principal components of the learned representation of the edges of the environment



(a) Episode reward in large square environment for target location 1

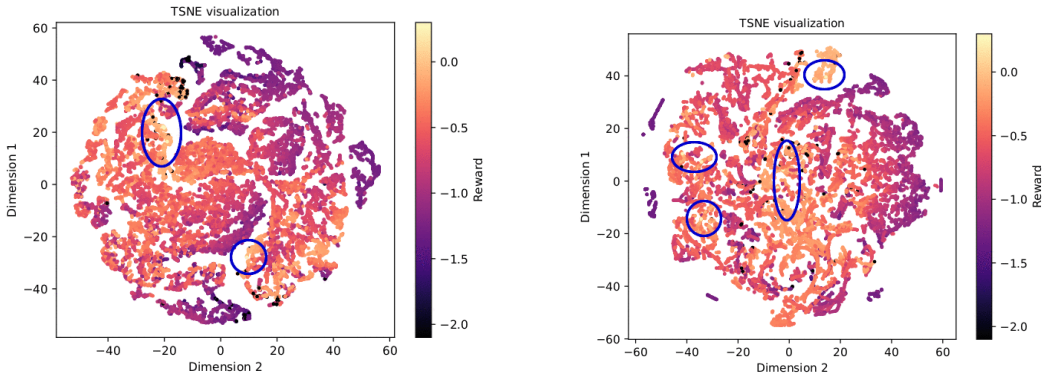
(b) Episode reward in large square environment for target location 2

Figure 5.22: Episode reward for both the recurrent and non-recurrent state encoder for two target locations in the large environment

experiments have been performed. To test how well each of the models generalise to unseen data, a new set of test data is gathered. Instead of gathering this data using the random policy, the data is gathered by training an RL policy, as this will ensure that the newly gathered data has a different distribution than the training data. More information on the new dataset can be found in the appendix E.1.

Figures 5.23a and 5.23b show the resulting t-SNE plots for the non-recurrent and recurrent state encoders respectively. The blue circles indicate the places where the target location has been plotted. Firstly, it can be seen that while the non-recurrent SR encoder maps all target location points to two locations. The recurrent state encoder however, maps the target location to four different areas in the plot. Generally speaking, it can be seen that the non-recurrent SR encoder preserves the color gradient better than the recurrent case. The t-Sne visualisations of both encoders when mapping the training data to the states, seen in Figures 5.19a and 5.19b for the non-recurrent and recurrent encoders respectively, show that on the training data, the recurrent SR is better. This shows that the non-recurrent state encoder network generalises better to unseen data.

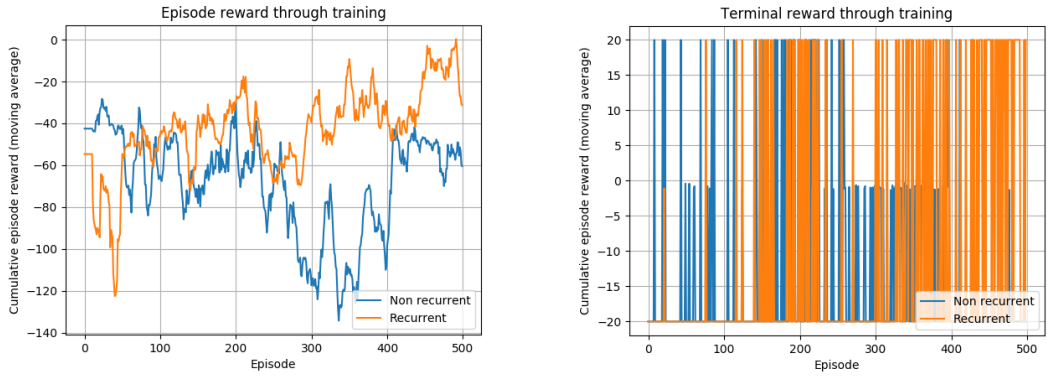
A correlation analysis shows that the recurrent encoder network has worse generalisation performance than the non-recurrent state encoder. This is mainly seen in the x component, the correlation with x is very low in the test data using the recurrent state encoder. The non-recurrent state encoder also shows much lower correlation in the test data, but not as severely



(a) T-SNE visualisation of the SR using the non-recurrent state encoder

(b) T-SNE visualisation of the SR using the recurrent state encoder

Figure 5.23: T-SNE visualisation of the state encoder applied to test data



(a) Cumulative reward in the single colored L-shape environment

(b) Terminal reward in the single colored L-shape environment

Figure 5.24: Episode and terminal reward signals in the L-shaped environment

as the recurrent state encoder. Further analysis of the Q-value and the correlation matrixes between the learned representation can be found in the appendix E.

5.3.2 L-shaped environment with same color walls

The L-shaped environment is adapted to be more challenging. This is achieved by changing the colors of the walls to all have the same color. Again, the recurrent and non-recurrent models are tested and compared. In this environment, 500 training episodes are shown because that the LSTM model still showed improvement after 250 episodes.

Figure 5.24a shows the moving average of the episode reward throughout training for both the recurrent and non-recurrent model. From this graph can be seen that the recurrent model ends up having a higher score, but is not converging to reach the target each time. Figure 5.24b shows the terminal reward that is achieved for the recurrent and non-recurrent models in this environment. For the entire 500 episodes, the non-recurrent model reaches the target location 45 times, while the recurrent model reaches the target 117 times. In the last 100 episodes of training, where the noise added to the actions has a small magnitude, the difference becomes larger. The non-recurrent state representation policy reaches the target only two times in these 100 episodes. The recurrent based policy reaches the target location 50 out of 100 times.

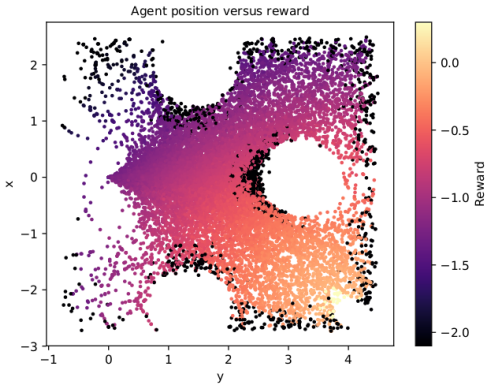


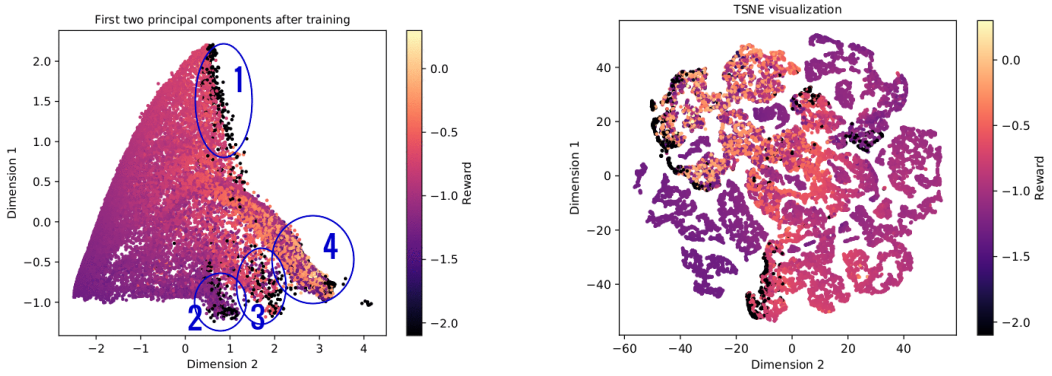
Figure 5.25: Dataset used for training the state representation on the large environment with cones

5.3.3 Square environment with colored cones

An even more challenging environment was created, shown in Figure 4.3b. This environment is challenging, because the walls of the environment are highly ambiguous, as these are the same color. The environment is square, adding symmetry to the environment. The colored cones can be used to locate the robot. However, to reach the target, there are many states in which the cones cannot be seen. To solve this environment, a relatively long sequence length is needed for training the recurrent network.

For reference, the robot locations in the training dataset are included, with the color indicating the reward that is received for being at that position. This plot also shows the target location. This can be found in Figure 5.25.

First, the state representation using the standard, non-recurrent framework is considered. The first two principal components of the learned representation can be found in Figure 5.26a, and the resulting t-SNE plot in Figure 5.26b.



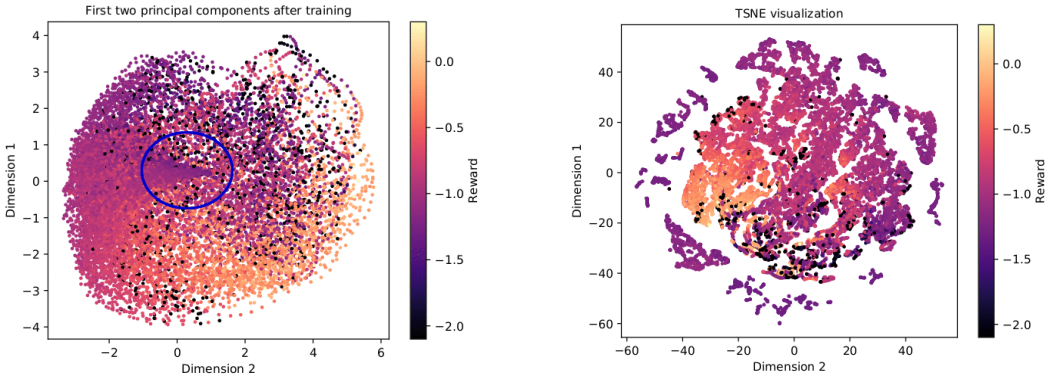
(a) First two principal components of the learned SR

(b) T-SNE visualisation of the learned SR

Figure 5.26: Visualisation of the learned SR of the large environment with cones using the non-recurrent state encoder network

Figure 5.26a has numbered ellipses drawn into the plot. In the plot can be seen that the three cones are represented in distinct parts of the representation, at ellipse one, two and three. Ellipse one shows the middle cone, ellipse two the cone farthest from the target, and cone three the other cone. These cones are represented very distinctly in the representation, and are not connected smoothly between those. Ellipse four encircles the part of the representation which represents the area of the target location and the area on the opposite of the middle cone. It can be seen that these two areas are mixed together, as colors are mixed in this area. This inability to separate those two areas can also be seen in the t-SNE plot of the learned representation, as

can be seen in Figure 5.26b. While part of the representation shows a smooth color gradient, parts of the representation shows mixing of colors. This signifies that the learned representation is not smooth and is not able to separate distinct areas of the environment. This state representation did not allow for learning a good policy.



(a) First two principal components of the learned SR

(b) T-SNE visualisation of the learned SR

Figure 5.27: Visualisation of the learned SR of the large environment with cones using the recurrent state encoder network

The plots of the first two principal components and the t-SNE visualisation of the learned representation using the recurrent model can be found in Figures 5.27a and 5.27b respectively. Judging from the t-SNE visualisation, the recurrent state representation learning framework is better able to separate the different areas in the representation. To achieve a good separation between the states, the sequence length for training the recurrent state representation network needed to be increased to 16.

The plot of the first two principal components highlights the position of the origin with the blue circle. This is an artifact of the fact that the first 15 states are not trained explicitly, but only used compute the 16th state. Going to longer sequence lengths aggravates this problem. This limits the sequence length to use only small sequences, which in turn limits the class of environments which can be solved using the recurrent state representation framework. Instead of ignoring the first states in a sequence, padding can be applied to also train the encoder network using the first states in the sequence. However, padding distorts the results, as the biases in the network will give a non-zero state that is transferred in time. Instead of padding the sequence with zeros as observations, a variable sequence length can be used during training. This however introduced artifacts into the state representation and did not give improved results. Using a shorter sequence length only to train the observations in the start of the sequence, such that all observations in the sequence are used for training the encoder network gave similar results. It did not show an improved state representation, and suffered similar artifacts in the state representation for the initial states.

Despite the recurrent encoder network producing a better separated state representation, no successful policy could be trained.

5.4 Additional experiments

This section covers some further experiments that have been performed. These experiments have been performed using the non-recurrent state encoder network, which is trained using prior set three.

5.4.1 Visual feature close to the target

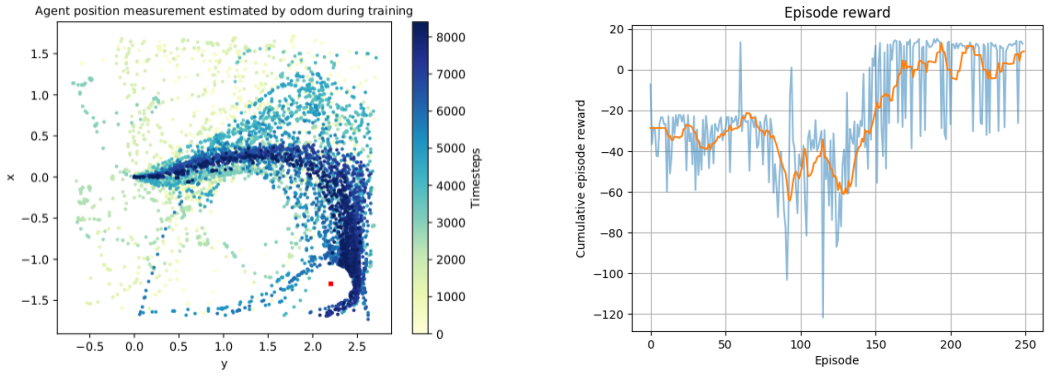
From earlier experimentation in the large square environment, it was found that target location two, close to one of the walls was hard to reach 5.22b. From plotting the first two principal components of the learned representation of observations close to the walls it was found that these points close to the walls were clustered together 5.21a. To test whether performance would improve if a visual aid was added close to the target that would help to localise the robot, the large square environment was adapted. A visual feature was added on the wall close to the target location, the environment can be seen in Figure C.1a.

Figure 5.28: Moving average of the episodic reward in the large square environment comparing adding a visual feature close to the target versus no visual feature close to the target

Figure 5.28 shows the episodic reward during training for both the environment with a visual feature added as the environment without visual feature. From this plot it can be seen that the performances of both agents are relatively close. In the last 100 episodes, in the environment with visual feature reaches the target 80 times, versus 56 times for the standard environment. As previously mentioned, in the standard large square environment, the agent learns to reach the target location in 250 episodes one out of three training runs. With the visual feature added, the agent learns to reach the target reliably across three runs.

5.4.2 Obstacle

So far, most environments have been open spaces without obstacles. To test whether the state representation can learn to avoid an obstacle, an obstacle is placed between the starting position of the robot and the target location in a square environment. An overview of the environment can be found in Figure C.1b.



(a) Robot positions during training

(b) Episode reward during training

Figure 5.29: Results training a policy to reach the target location in the environment with an obstacle

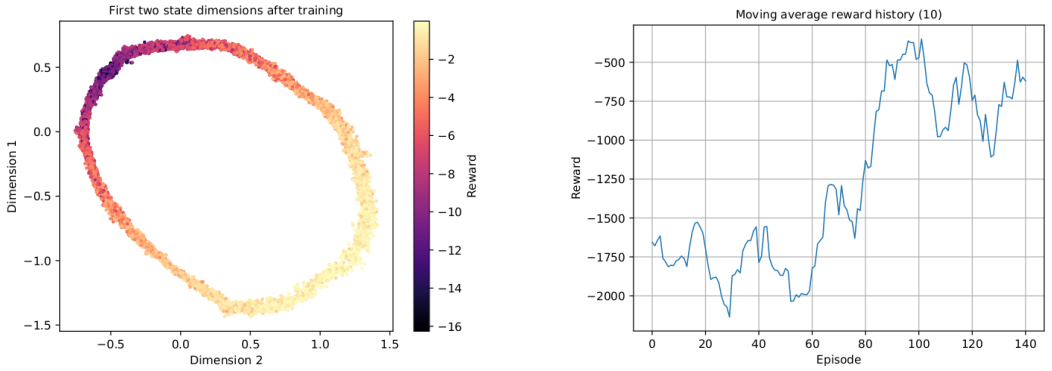
Figure 5.29a shows the x and y positions of the robot during training. Figure 5.29b shows the episodic reward during training. From this it can be seen that the agent learns to reach the target reliably. The agent learns to avoid the obstacle.

5.4.3 Gym environments

To test the robotic priors in different types of environments and different tasks, the priors are used to train a state representation framework in two gym environments. As the gym environments only have an rgb input, the network architecture is adapted. The architecture can be found in Figure 4.6. The input dimensions was set to [100,100,3].

Pendulum

The first gym environment in which the robotic priors are tested is the pendulum (37). The agent has to swing the pendulum upright, and then balance the pendulum. The reward is based on the position of the pendulum, the speed and the magnitude of the input torque. To train the state representation, 18000 observations are collected. The state representation is trained using the same priors, using the same hyper-parameters. The resulting state representation can be found in Figure 5.30a. From this figure can be seen that the states lie on a circle, as is expected.



(a) Learned SR in the pendulum environment

(b) Episode reward from training in the pendulum environment

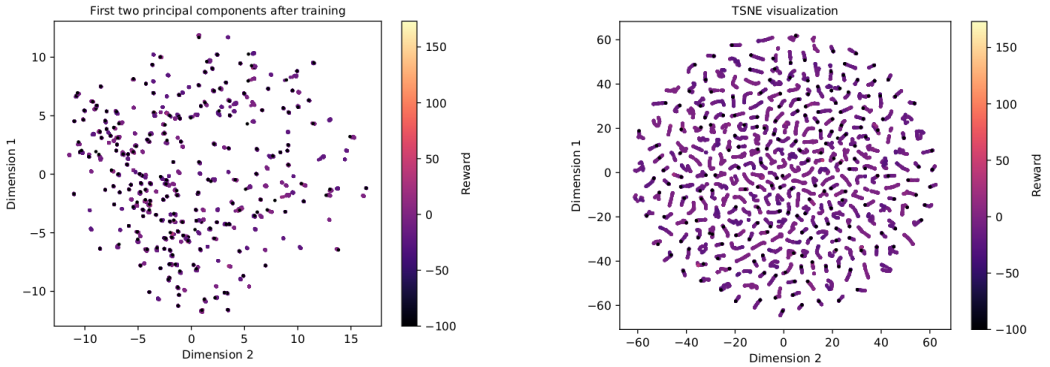
Figure 5.30: Results in the pendulum environment

Furthermore, it can be seen that the reward changes relatively smoothly, with a clear color gradient in the states. Although not solely determined by the position, the reward signal is still dominated by the angle of the pendulum.

From the plot of the learned state representation can be seen that the encoder network learns to encode the x and y positions of the pendulum end effector. This is not sufficient to solve this environment, as the speed of the pendulum is necessary to balance the pendulum. Therefore the state used as input for the DDPG algorithm is set to $[\hat{s}_t \hat{v}_t]$, where $\hat{v}_t = \hat{s}_t - \hat{s}_{t-1}$. At $t = 0$ $\hat{v}_t$ is set to zeros.

Figure 5.30b shows the episodic reward throughout training. After episode 90 it learns to balance the pendulum in most episodes. Because the agent is not able to balance the pendulum exactly in the middle, thus having to apply more torque on the pendulum to balance it, the episode reward is low compared to training on ground truth data. Using the ground truth data, the policy converges after episode 40, after which it achieves mean episode reward of approximately -144.

This task benefits from having a accurate state representation, however, the maximum accuracy is limited by the resolution of the observation. Before using observations of 100x100 pixels, observations of 30x30 pixels were used. Although the learned SR was similar, this did not allow



(a) First two principal components of the learned SR

(b) T-SNE visualisation of the learned SR

Figure 5.31: Learned SR in the LunarLander environment

the agent to learn to balance the pendulum. Increasing the resolution of the observation further will likely reduce the gap in episode reward between the policies trained on the ground truth states and the state estimations.

LunarLander

The second environment in which the robotic priors are tested is in the LunarLander-continuous (38) gym environment. The target of this environment is to gently land a moon-lander on a landing pad by controlling the thrust of three engines. The landscape in the environment changes slightly for each episode. This is different from all previous tasks, where the starting state was equal between episodes. The state encoder network is trained using 25500 observations. The resulting representation can be found in Figure 5.31. From the plots can be seen that the state representation is not one smooth field, but instead many small clusters of points. A similar phenomenon was observed by (31), where the different placement of a distractor object caused different clusters in the learned state representation. In this environment, the random landscape causes each episode to end up with it own cluster. In each cluster one can find a terminal state, distinct from other states by having a lower reward. This clustered state representation did not allow the agent to learn the task successfully.

5.5 Changing encoder architecture

This chapter considers some final experiments and late results that could not be included in the main body of this work.

5.5.1 Architecture

As mention section 5.3, the states close to the walls of the large square environment are not represented as well as the other states. This can be seen from the clustering of the points near to each of the walls, where the real distribution would be more even. Close to the walls, the camera observation does not give as much information as it does for other points, as the camera image will be identical from different points in the environments. However, the observation from the lidar sensor should be sufficient to disambiguate these states.

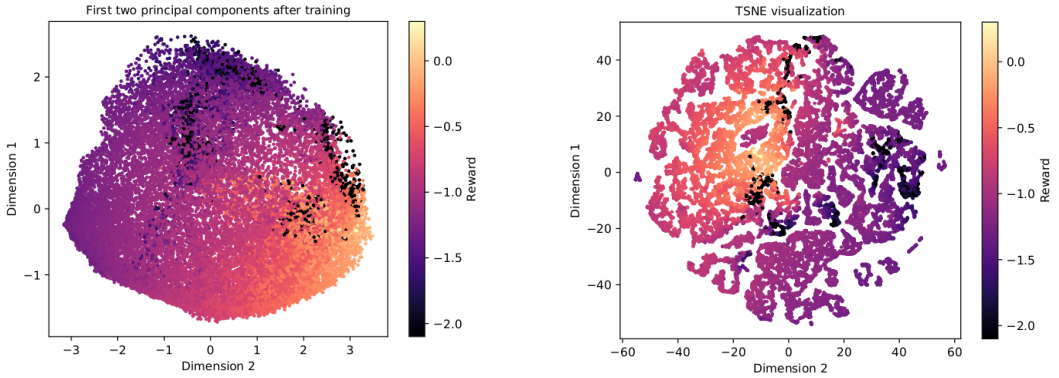
In the final stages of this work, it was found this was caused by the architecture of the encoder network. This architecture, which can be found in Figure 4.5, fuses the two streams from the camera and lidar at the end of the network. After fusing the two streams, there is only a linear layer in the network. This limits the network to combine the information from both streams linearly. This can be an issue, especially apparent close to the walls, as close to the walls the camera data does not give as much information is it does when the camera can see multiple walls in the image.

To rectify this issue, the existing architecture was altered. The new architecture is defined in Figure 4.7. This new encoder network architecture is used in the experiments shown below.

5.5.2 Initial results

This architecture was first tested in the large square environment, as this environment shows whether the new architecture will be better at disambiguating positions close to the walls. After learning the state representation for the large square environment, an RL agent was trained to reach target location two. This target location was difficult to reach using the previously learned state representation. It was hypothesised that this was due to the bad representation close to the walls. The new state representation should be good enough for an RL agent to learn to reach positions close to one of the walls, as this would show the new architecture provides an improvement.

To learn a good state representation which allowed for learning to reach the target location, some tuning of the hyper-parameters was required. The magnitude of the L2 regularization was reduced to 10^{-6}. The prior weights were adjusted to $L_{\text{priors}} = 1 * L_{\text{temp}} + 2 * L_{\text{caus}} + 1 * L_{\text{prop}} + 1 * L_{\text{rep}}$. The resulting state representation can be found in Figure 5.32. Comparing the first two principal components of the learned representation using the new architecture with the learned representation using the old architecture as found in Figure 5.18a shows some clear differences. The old architecture shows the walls as clusters, barely connected. The new architecture shows that the walls are connected, and shows a smooth transition between the walls of the environment. The state estimates close to the walls of the environment are plotted in Figure 5.34a. This shows how well the states close to the walls are represented. From this plot shows that the clustering of points close to the walls is reduced, and the rewards are more smoothly connected compared to the old architecture.



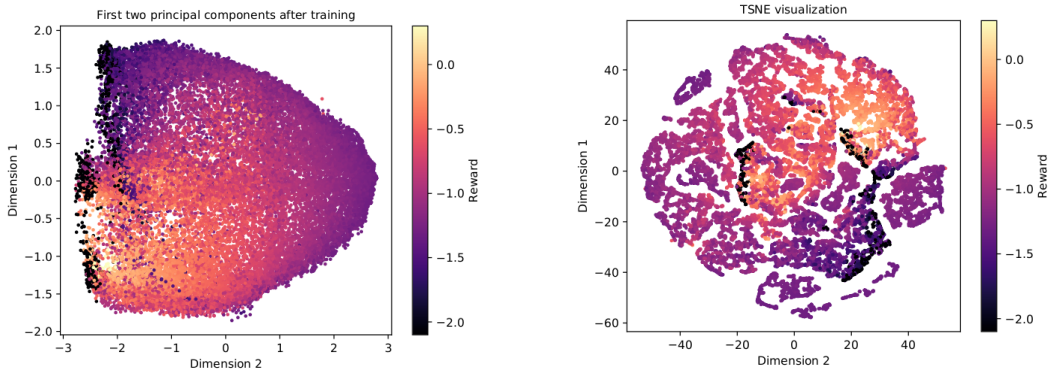
(a) First two principal components of the learned SR

(b) T-SNE visualisation of the learned SR

Figure 5.32: Learned SR of the large square environment using the new encoder network architecture

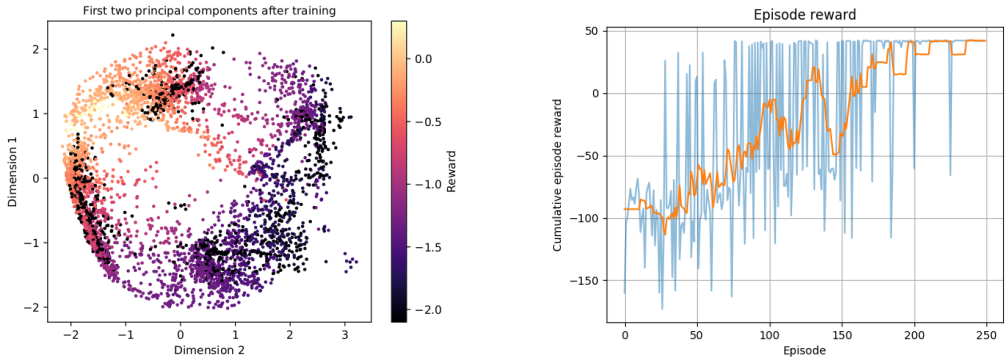
For the sake of comparison the learned representation using the old architecture is plotted again, see Figure 5.33. Comparing the first two principal components shows a clear difference. The old architecture produces a state representation that less smoothly connected. Also, the walls are not connected, but instead clustered. The new architecture shows that the walls are connected better. The t-SNE visualisations also show that the new architecture is an improvement. The t-SNE visualisation of the new architecture, shown in figure 5.32b shows a clearer color gradient towards the target location.

To test whether the new state representation is indeed better using the new encoder network architecture, the learned state representation will be used to train an agent. Target location two will be used for this, as this target could not be reached reliably based on the previous encoder architecture. Figure 5.34b shows the episode reward throughout training based on the new state representation.



(a) First two principal components of the learned SR **(b)** T-SNE visualisation of the learned SR

Figure 5.33: Learned SR of the large square environment using the old encoder network architecture



(a) First two principal components of the learned SR, showing the states close to the edges of the environment as in 5.20 **(b)** Episodic reward for target 2 in the large square environment

Figure 5.34

This shows that the new architecture gives an improved state representation as this target location can be reached reliably using this new state representation.

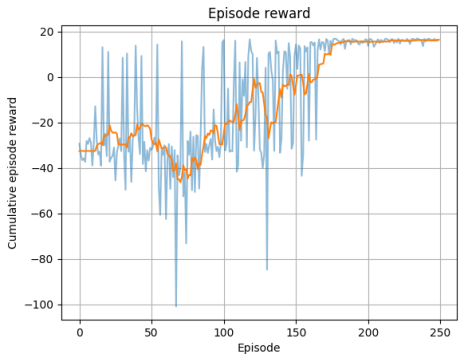
5.5.3 Testing in other environments

It was found that using the new state encoder architecture allowed for learning to reach target location two in the large square environment. However, this new architecture should also at least be able to solve all previously solved environments. Therefore, each of these environments are tested again using the new encoder architecture.

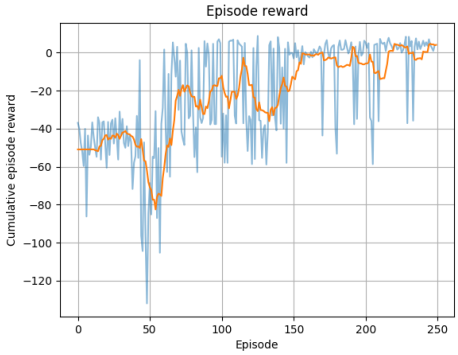
Figures 5.35 and 5.36 show the training results for the environments and targets that were successfully solved using the old architecture. Only the large square environment shows slightly unreliable performance. In the rest of the environments, learning still happens as expected.

After validating that the new encoder architecture and hyper-parameters allow for learning successful policies in the environments in which the old architecture was also successful, other environments can be considered. The first environment in which the old encoder network did not allow for learning a good policy, is the L-shaped environment with all its walls the same color.

Figure 5.37 shows the episode reward achieved during training in the L-shaped environment with the same colored walls. In the last 100 episodes, the robot reaches the target 51 times, of which 33 times in the last 50 episodes. The new architecture does improve over the old encoder architecture, as the policy now learns to reach the target location successfully. However,

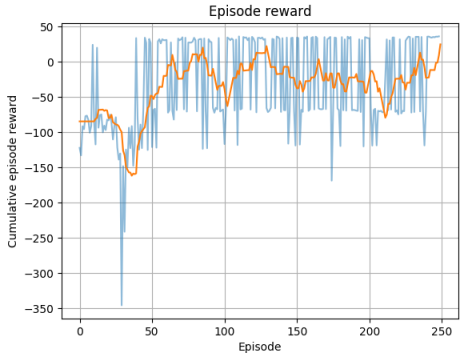


(a) Episode reward for the small square environment



(b) Episode reward for the L-shaped environment

Figure 5.35: Episode reward using the new encoder network architecture for the small square and L-shaped environment



(a) Episode reward for the large square environment, target location one

(b) Episode reward for the environment with obstacle, as can be found in Figure C.1b

Figure 5.36: Episode reward using the new encoder network for the large square environment and the obstacle environment

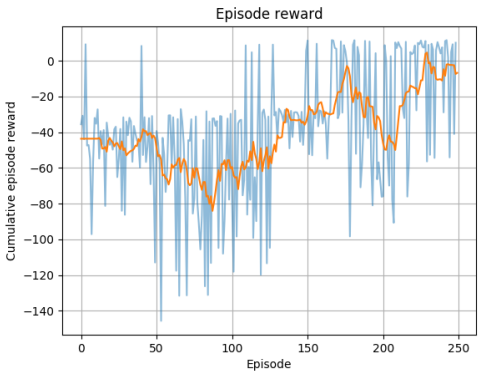
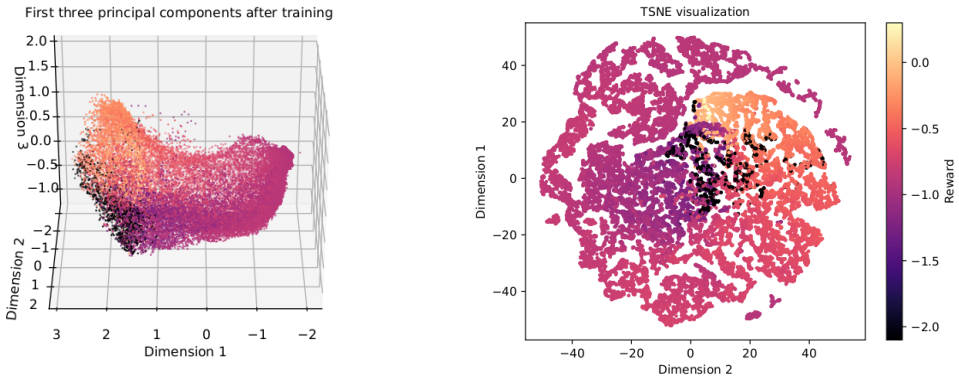


Figure 5.37: Episode reward using the new encoder architecture in the L-shaped with the same colored walls

it is still not reliably reaching the target location, indicating that even with this new encoder architecture, the learned state representation is not good enough. Figure 5.38 shows the learned representation in the L-shaped environment with same colored walls using the new encoder architecture. To better visualise the representation, the first three principal components are plotted, as this gives a better visualisation in this particular case. What can be seen from the visualisations is that the learned state representation fails to capture the physical structure of the environment well. While the learned representation shows a good color gradient, the locations close to the starting position are mapped closely to the positions of the target location. A simi-



(a) First three principal components of the learned SR

(b) T-SNE visualisation of the learned SR

Figure 5.38: Visualisation of the learned SR in the L-shaped environment with the same color walls using the new encoder architecture

lar effect was seen previously, as described in 5.2.1. This shows that the distance between states in the learned representation is not representative of the difference in reward. The robotic priors do no directly account of the difference in reward between observations and does therefore not account for the space in which there are no observations.

6 Conclusions and recommendations

6.1 Answering the research questions

The field of reinforcement learning shows great potential in advancing the field of robotics. This has the promise of making robots smarter and tackle more complex tasks. To achieve these tasks, robots must often rely on generic sensors, as these are cheap and can be used to tackle a wide verity of tasks. These generic sensors often produce high dimensional data, which makes learning tasks using RL more challenging and requires more data for training agents. This is also true for mobile robot navigation, the problem on which this work has focused. State representation learning aims to solve this problem of high dimensions of the observation space by learning a low dimensional state representation. This state representation maps the high dimensional observations to a low dimensional state space. This learned state representation is then used for training an RL policy. In the context of robotics, learning such a state representation can be done using robotic priors. Robotic priors have been used for mobile robot navigation in literature, but not in a continuous action space. This leads to the first research question:

1 How can robotic priors be used to learn a good state representation with a continuous action space?

Section 3.1 has presented the two sub-questions which are used for answering this research question:

1a Are the proposed priors from (24) and (11) able to learn a good representation when using a continuous action space, when these are adapted to better work with continuous actions?

1b Can the current sets of priors be adapted to better make use of the continuous nature of both the reward signal and action space by removing conditioning on the training points?

To answer these questions, several environments were created. These environments varied in size and difficulty. For each questions, two sets of robotic priors were created, as presented in 4.1. The first two sets of robotic priors were adapted from priors found in literature. Two other sets of priors were created, in which the conditions were replaced with continuous expressions. This is a novel approach to implementing robotic priors. To answer the sub-questions, and consequently research question 1, each set of priors was tested. To test each set of priors, each set was tested in the three environments presented in the section 4.1.

Using each set of priors, a state representation was learned using a dataset gathered from each environment. To judge the quality of the learned representation, several methods were used. The first method was to qualitatively analyse the learned state representation. For this analysis, the first two principal components to the learned state representation and a t-SNE visualisation were plotted. One way to analyse a learned representation is to look at the color gradient present in both plots. These plots should show a smooth color gradient, as this means that different areas of the environment are separated well.

The most important way of evaluating a learned state representation is to train a policy which is based on this state representation. This is most important, as the goal of learning a state representation is to more easily learn to solve a task, which in this work entails navigating the environment to reach the target location.

Using the results in section 5.2.1 research question 1a can be answered. It was found that adapting the priors from the work of (24), called prior set one, was not able to learn a good representation in two of the three environments. The priors in set two, which were priors adapted from (11), were able to learn a good representation in the small square environment and the L-shaped environment. In both environments, a good policy could be learned that learned to reach the target location. This set of priors did not learn a good representation in the large square environment. The hyper-parameters for both sets priors were kept the same for each environment, given that these priors work well in the first two environments, it is possible that a good representation could be learned for each of the environments if the hyper-parameters are tuned appropriately for each of the environments, although this has not been achieved in this work. This sensitivity to the hyper-parameters makes these sets of robotic priors less suitable, as this means spending more effort to successfully solve an environment.

The results presented in section 5.2.2 can answer research question 1b. It was found that using the causality prior as designed initially in 3.14 did not result in a good state representation. Therefore, the causality prior was adapted 5.1. Using this new causality prior in both prior set three and four showed good results for each of the environments. Both sets of priors worked equally well. These priors were able to learn a good representation in each of the environments without the need of extensive hyper-parameter tuning. This can be a big advantage, as tuning these parameters can be an expensive exercise. Furthermore, the implementation of these continuous priors from set three and four is significantly easier than the conditional priors from set one and two. This is evident when comparing the implementation details in section 4.2.2. It was further shown that the priors in set three is also able navigate an environment with an obstacle, and was successful in learning a state representation in the gym environment pendulum.

After performing all the experiments, the research question can be answered. Two methods for learning a state representation with a continuous action space have been presented. The first method is to relax the conditions in the conditional priors and the second approach is to replace the conditions with continuous expressions. Both approaches have been shown to be viable. It was shown that relaxing the conditions on the priors made these priors sensitive to the hyper-parameters of these priors. The continuous priors was found to work better as these were not as sensitive to hyper-parameter tuning and worked well in the different environment without requiring further tuning.

To answer the second research question, further experiments have been performed. The second research question:

2 To what extend can a recurrent state representation be learned using the adapted priors in a continuous action space?

It was found that using a recurrent encoder did help to distinguish ambiguous states, especially when the colors of the walls could not be used to uniquely distinguish the states. Learning a policy using the recurrent state encoder showed that the recurrent state encoder did not allow to learn a policy that reaches the target location reliably. It was found that the recurrent encoder generalises less well to the distribution of data that is encountered when training a policy. In the L-shaped environment with all walls the same color it was found that the non-recurrent state encoder did not allow for learning a policy to reach the target with the old encoder architecture, while the recurrent version did, albeit in only half of the episodes. In the environment with colored cones, no good policy could be learned. The recurrent and non-recurrent state encoders have been compared extensively. It was found that the recurrent state encoder network has poorer generalising performance from training to test data. This lack of generalising performance limits the practical usefulness of the recurrent state encoder, as this does not allow for learning a good policy that reaches the target reliably. Furthermore, in the partially ob-

servable environment with the colored cones, no good state representation could be learned. When long sequences are required for training the state representation, the quality of the state representation suffers. Whether the recurrent state encoder network can learn a good state representation using the current priors in partially observable environments remains an open question.

It was found that the architecture of the state encoder network can play a significant role in the performance of the reinforcement learning algorithm. The architecture of the non-recurrent state encoder network was adapted and tested and showed improvement over the previously used architecture. Due to time constraints, the recurrent state encoder network architecture has not been adapted. It is possible that adapting the architecture of the recurrent encoder network will improve the results that have been achieved.

6.2 Recommendations for future work

This work has focused on applying state representation learning with robotic priors to mobile robot navigation. Recommendations for further work on the topic of state representation learning using robotic priors are given below.

6.2.1 Exploration

Exploring the environment is crucial for learning a good state representation. In this work, the used environments could all be explored sufficiently using only OU noise. If environments become more complex to navigate, by having more bends and tighter spaces, using only OU noise will not give a sufficiently good exploration of the environment. Harder to reach spaces will hardly be explored by using the OU noise. To explore these harder to reach spaces, another approach is be needed. To achieve good exploration some form of intrinsic reward could be used which encourages the robot to explore unseen parts of the environment.

6.2.2 Distribution of training data

Having a good distribution of training data across the environment is important for learning a high quality state representation. This is especially true if the state representation is used to reach arbitrary target locations within the environment. Solving the problem of exploration could help to ensure having a good distribution. However, the SRL memory size is limited and could overflow during exploration. The naive method which is used in this work is to remove the oldest experiences. This however does not ensure a good distribution of states across the environment as some parts of the environments might be explored in the beginning of training, and others later. Choosing which experience to remove from the SRL memory when the memory is overflowing can help to ensure a good distribution.

6.2.3 Environment dynamics

This work has only considered the problem of mobile robot navigation in a limited set of environments. In this problem, a velocity set-point is set by the policy. This velocity set-point is then converted to wheel velocities by the internal controller. This command is then executed for a set duration, making the steps similar in time. The SRL method has no need to learn the velocity of the robot as this is set by the controller. Therefore, the dynamics of the environments were not important for solving the task. In other robotics applications this can be different. Often the dynamics are important for solving tasks. Expanding the concept of state representation learning to the class of problems where the environment dynamics are important for solving the task can expand the class of problems that can be solved using SRL.

6.2.4 Changing environments

This work has focused on a class of environments that stay static between episodes. In a real world situation, this will rarely ever be the case, as environments will change over time by different lighting conditions or objects moving about. The current set of robotic priors are not able to distinguish between sources of variation that are relevant for the task and sources that are not relevant. Furthermore, if changes happen between episodes but not within episodes, the learned state space will no longer be coherent, as further detailed in section 5.4.3. To improve applicability of the state representation learning using robotic priors framework, this would need further work.

6.2.5 Partially observable environments

This work has performed further investigation into the recurrent state representation learning using robotic priors framework. It was found that a recurrent state encoder can be trained using the same robotic priors as are used to train a non-recurrent state encoder network. However, in this work, no good representation could be learned in the environment with colored cones, which is partially observable. As mentioned previously, changing the architecture of the recurrent state encoder network could help to improve the results achieved in this work.

A Additional background information

A.1 Compression of observation

It can be assumed, given limited noise, that compressing the observation should retain most of the relevant information. Observations can also often be brought back to a lower dimension with minimal loss, such that the loss of information is minimal. This idea can be exploited for learning a state representation.

One method of performing dimensionality reduction is Principal Component Analysis (PCA). PCA is a linear transformation, which is able to compress and decompress observations with minimal reconstruction error (14). This dimensionality technique was employed to play a Super Mario game (12), where only 4 dimensions were needed to obtain a similar performance to using the observations directly. It was further employed for robotics simulation to solve the Swimmers and Mountain Car simulations (40).

This linear dimensionality reduction can be expanded to non-linear dimensionality reduction methods using auto-encoders (21). The state representation is learned using samples from the environment, which trains both the encoder and decoder to map the input to a low dimensional state, and to project it back to its original input. After these mappings have been learned, the encoding part of the network can be used to map new observations to a lower dimensional state, which can be used as input to the RL algorithm. The well known variants on the AE, the variational autoencoder and the denoising autoencoder can also be used for learning a state representation (52; 22).

These models map only the current observation to the state, and as such cannot model any dynamics. Furthermore, auto-encoders are known to sometimes ignore smaller objects in the input, which can be significant if these are relevant for the task. Furthermore, the auto-encoder framework does not distinguish between environment elements based on whether these are task relevant and irrelevant. It will model factors of variation in the data, which would mean that it will model elements of the environment which are not relevant for the task.

The AE framework can be expanded such that environment dynamics will help constrain the state space. (16) adds the constraint that the transition between two consecutive states s_t and s_{t+1} must be linear.

A.2 Dynamics model

Another class of methods aims a modeling the environment dynamics from observations.

A.2.1 Forward model

The forward model aims at forcing the states to encode all information necessary to predict the next state, given the action that was taken. This method works by encoding the observation to a state estimation, $\hat{s}_t = \phi(o_t)$, the next state is then predicted using the current state estimation and the action taken: $\hat{s}^p_{t+1} = \xi(\hat{s}_t, a_t)$. The model is then optimized to minimize $|\hat{s}^p_{t+1} - \phi(o_{t+1})|$.

Further constraints can be made to improve the representation. One common constraint is that the transition between states needs to be linear (16; 52), such that the next state is a linear combination of the current estimated state s_t and the action a_t.

Forward models are able to learn representations of controllable factors in the environment. The model will model those aspects of the environment which can be affected by the action the agent takes. Therefore, it will model elements of the environment which are predictable. Environment aspects that cannot be controlled by the actions of the agent, but instead are controlled by other factors are often not predictable, and thus these will not be encoded into

the state. Forward models will thus model predictable aspects of the environment into its state, and will tend to not model other environmental aspects.

The forward model can be combined with the auto-encoder framework to improve the state representation and stability (9; 26). A forward model combined with the AE framework can be implemented the following way. Using the estimated state s_t and a_t, the next state $\hat{s}_{t+1}$ can be predicted. $\hat{s}_{t+1}$ can then be mapped to $\hat{o}_{t+1}$. The model is trained to minimise the pixel-wise loss between $\hat{o}_{t+1}$ and o_{t+1}.

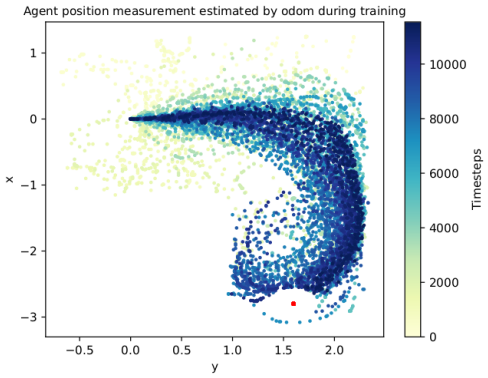
A.2.2 Inverse model

Instead of a forward model, in which a future state is predicted, an inverse model can be employed. The inverse model first projects observations o_t and o_{t+1} onto states $\hat{s}_t$ and $\hat{s}_{t+1}$ and then tries to predict the action $\hat{a}_t$ that explains the transition from o_t to o_{t+1}. Just like the forward model, this poses constraints on the state representation, in the case of the inverse model to make the action predictable from the states.

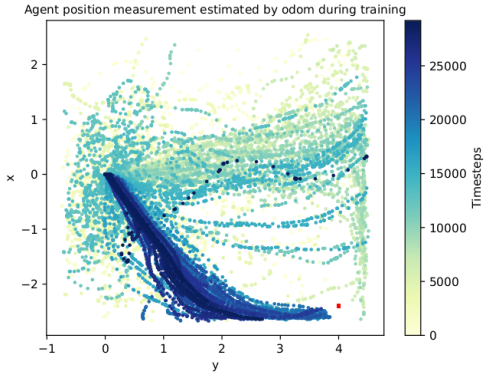
The forward and the inverse model can be combined, and are found to be complementary. (7) has used the combination of a forward model and an inverse model to learn the state representation for a robotic arm. The robotic arm was able to learn to poke objects. In their work it was found that using a forward model in combination with the inverse model regularizes the learned state space. Using the combination of forward and inverse model consistently outperformed the inverse model on its own.

The work of (13) combines the forward and inverse models with an auto-encoder, as well as the prediction of the instantaneous reward. Using all these modalities they were able to learn a state representation which gave relevant information for racing a virtual car on a racetrack.

B Additional figures

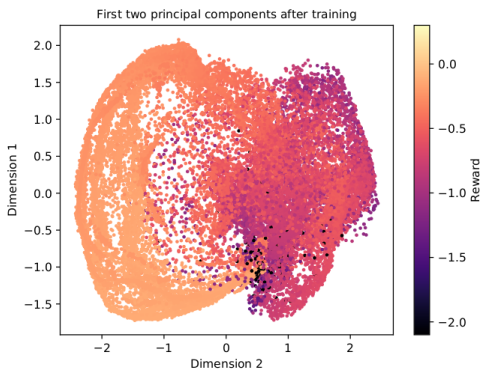


(a) Robot locations during training in the L-shaped environment

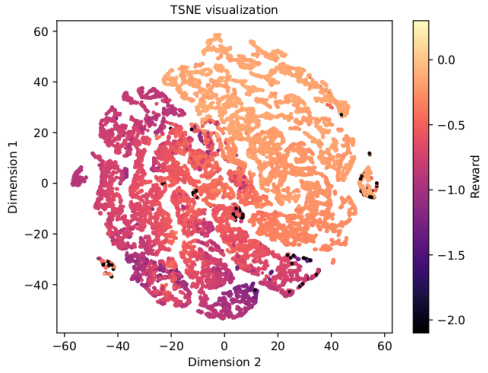


(b) Robot locations during training in the large square environment

Figure B.1: Paths taken during taken in the L-shaped and large environments while using SR based on prior set two

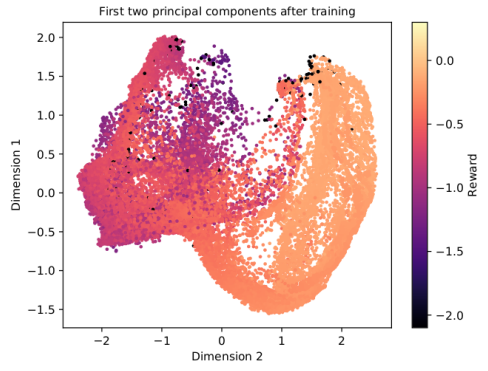


(a) First two principal components of the learned SR of the L-shaped environment using prior set one

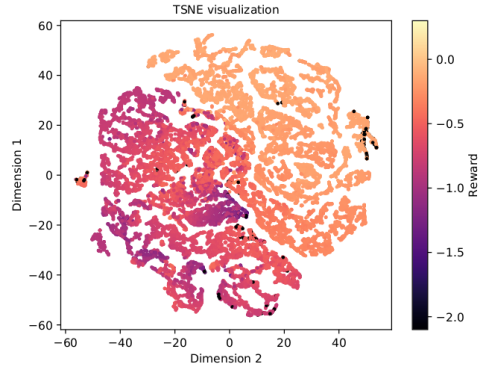


(b) t-SNE visualisation of the learned SR of the L-shaped environment using prior set one

Figure B.2: t-SNE visualisations of the learned state representation in the L-shaped environment using prior set one

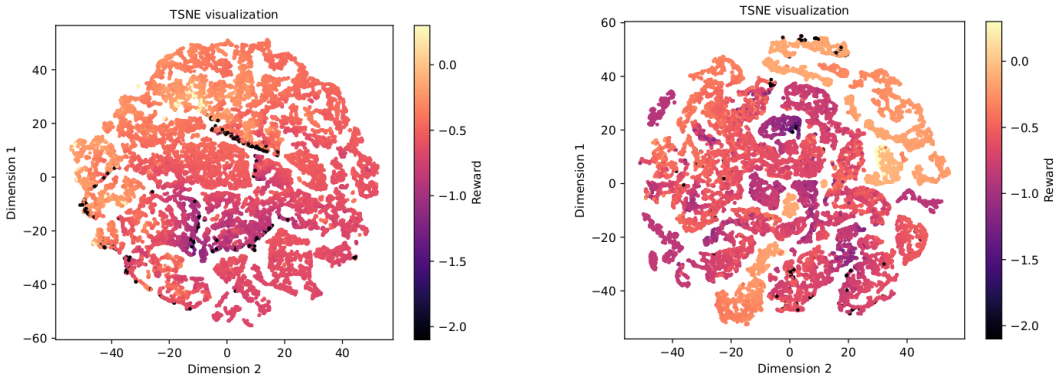


(a) First two principal components of the learned SR of the L-shaped environment using prior set tow



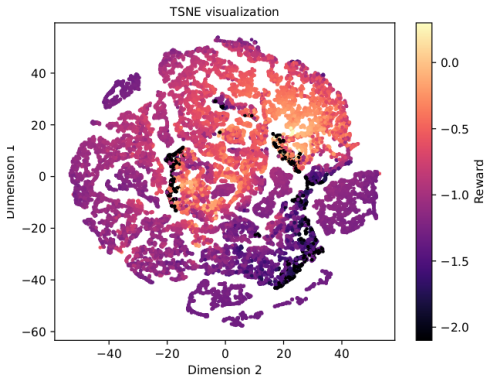
(b) t-SNE visualisation of the learned SR of the L-shaped environment using prior set two

Figure B.3: t-SNE visualisations of the learned state representation in the L-shaped environment using prior set two



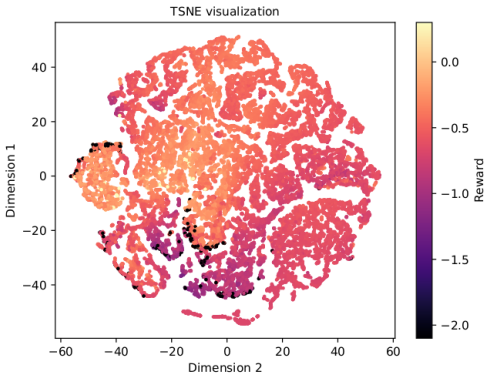
(a) t-SNE visualisation of the learned SR of the small environment using prior set three

(b) t-SNE visualisation of the learned SR of the L-shaped environment using prior set three

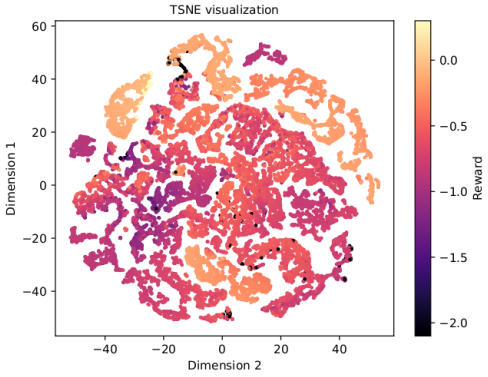


(c) t-SNE visualisation of the learned SR of the large environment using prior set three

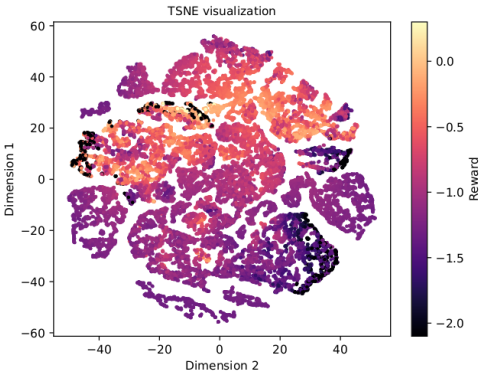
Figure B.4: t-SNE visualisations of the learned state representations using prior set three



(a) t-SNE visualisation of the learned SR of the small environment using prior set four



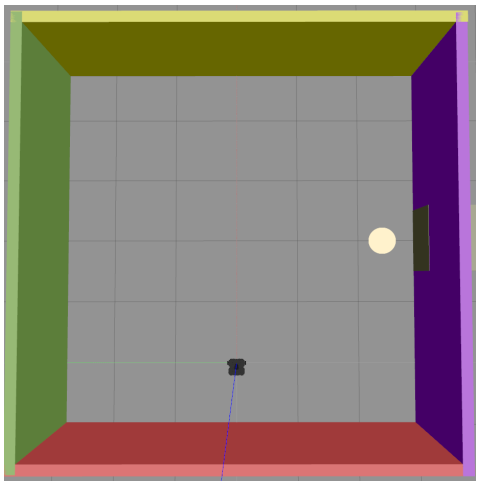
(b) t-SNE visualisation of the learned SR of the L-shaped environment using prior set four



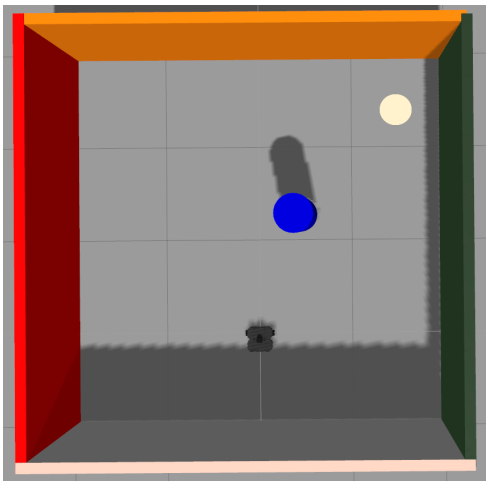
(c) t-SNE visualisation of the learned SR of the large environment using prior set four

Figure B.5: t-SNE visualisations of the learned state representations using prior set four

C Additional environments



(a) Overview showing the large square environment as in 4.2 with an added visual feature close to the target



(b) Overview of the square environment with an obstacle blocking the direct path to the target

Figure C.1: Overview of environments used in additional experiments, showing the initial position of the robot and the target location

D Removing the conditions from priors

Section 3.2.1 has covered an extensive analysis of the robotic priors which will be used in this work. This section has introduced the sets of continuous priors which is used in prior sets three and four. In these priors, the condition $a_{t_1} = a_{t_2}$ is replaced with the expression $e^{-\beta\|a_{t_1}-a_{t_2}\|^2}$ in the loss function. To see if using this expression in the loss function has any benefit, a set of priors is used in which the conditions of the original priors are removed and not replaced with an expression in the loss function. These priors then become:

$$L_{temp}(D,\phi) = \mathbf{E}\left[\left(\|\Delta s_t\|\right)^2\right] \tag{D.1}$$

$$L_{prop}(D,\phi) = \mathbf{E}\left[\left(\|\Delta s_{t_2}\| - \|\Delta s_{t_1}\|\right)^2\right] \tag{D.2}$$

$$L_{rep}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2}\|\Delta\hat{s}_{t_2} - \Delta\hat{s}_{t_1}\|^2\right] \tag{D.3}$$

$$L_{caus}(D,\phi) = \mathbf{E}\left[e^{-\|\hat{s}_{t_2}-\hat{s}_{t_1}\|^2}\right] \tag{D.4}$$

Figure D.1 shows the episode reward throughout training for small and large square environments and the L-shaped environment. The figure shows that in the small square environment, the policy converges to a good solution. In the large square environment, the policy learns to reach the target momentarily but it is not stable and not able to recover. In the L-shaped environment, no solution is found.

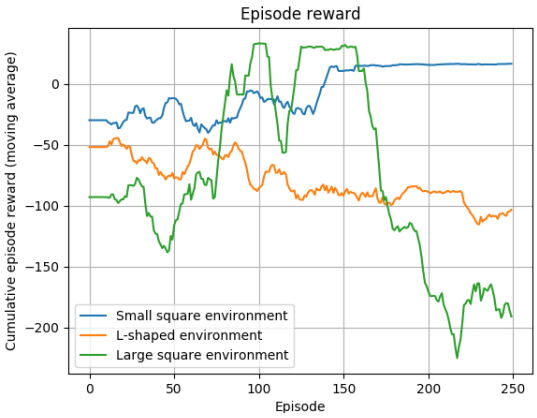


Figure D.1: Test dataset

E Generalisation of the state representation

E.1 Dataset for testing generalising performance

To test the generalising performance of the recurrent and non-recurrent state encoder networks, a test dataset was gathered. This was done by training a policy based on a trained recurrent encoder network. This network was not trained for 30 epochs but instead for 10 epochs. This made the learned state representation not good enough to learn a policy that would learn to reach the target reliably. This ensured sufficient exploration of the environment.

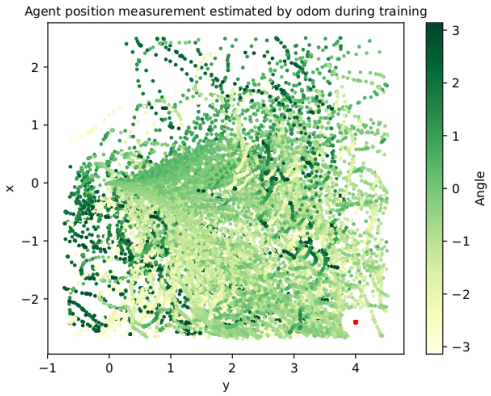


Figure E.1: Test dataset, used to analyse the generalising performance

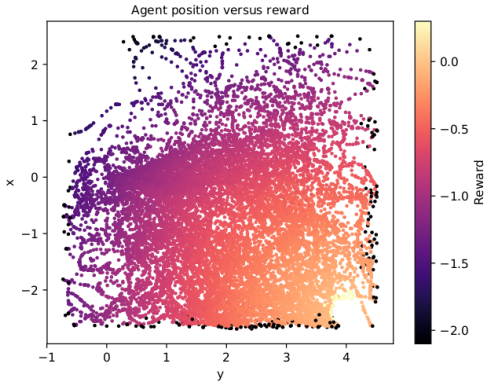


Figure E.2: Test dataset, where the locations are plotted versus the reward

Gathering the data using an RL policy instead of using random exploration ensures that the test data has a different distribution than the test data. This means that the actions are more varied, as well as the paths that are taken by the mobile robot. This test data is more aligned with the data that would be gathered when training a policy, and is therefore a good test of generalisation. The resulting dataset can be found in Figures E.1 and E.2.

E.2 Principal component analysis

As presented in section 5.1, the agent is able to successfully reach the target in each of the environments when training on the ground truth data. The ground truth data consists of five values, of which two represent the target location. The target location is not necessary for training the agent, as the goal is stationary. To train an agent based on a learned state representation, this state representation must learn to at least represent the x and y locations of the robot, as well as the orientation.

To test whether this is actually learned by the agent, the pearson correlation coefficient can be computed between the ground truth state, the x, y, and θ of the root and the principal components of the learned representation. It should be noted that this only evaluates the linear correlation between the ground truth state and the learned state representation, while this may be very non-linear.

This analysis is performed for both the recurrent state representation encoder and the non-recurrent version. Although this could be done for each of the environments, this analysis is performed only for the large square environment. Table E.1 shows the resulting correlation coefficients between the ground truth and the four principal components for the recurrent state representation encoder. Table E.2 shows the same for the non-recurrent encoder. The highest absolute correlation coefficient for each of the ground truth state components is made bold.

	PC 1	PC 2	PC 3	PC 4
x_r	**0.75993336**	-0.0751216	-0.356089	0.08148859
y_r	-0.00353692	**-0.82507391**	0.24372134	0.21590286
θ_r	-0.0591193	**-0.58792952**	0.10164354	0.46282379

Table E.1: Correlation analysis using the recurrent state encoder

	PC 1	PC 2	PC 3	PC 4
x_r	**-0.73927324**	0.01819131	-0.39570218	0.05921032
y_r	0.00736506	**-0.72162511**	-0.05094587	0.05244842
θ_r	0.01302098	**-0.76812087**	0.03203499	0.30170934

Table E.2: Correlation analysis using the non-recurrent state encoder

To test how well the state encoder network is able to generalise the learned state representation to unseen data, the same analysis can be done on the test data from Figure E.1. This could be considered a test of generalisation performance, although the non-linear nature of the state representation network makes these results more difficult to analyse.

	PC 1	PC 2	PC 3	PC 4
x_r	0.0779288	-0.1718507	**-0.18439722**	-0.04497203
y_r	-0.40031505	-0.22893654	-0.26159839	**0.45712542**
θ_r	-0.22440733	-0.34830969	-0.02868713	**0.44189775**

Table E.3: Correlation analysis using the recurrent state encoder on the data shown in Figure E.1

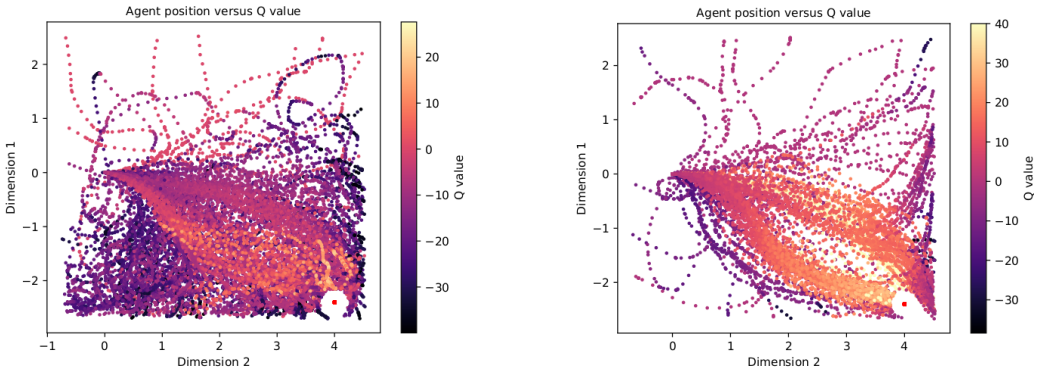
	PC 1	PC 2	PC 3	PC 4
x_r	-0.23461015	0.17487065	**0.435942838**	0.14831155
y_r	**0.4444722**	-0.3245973	-0.0497809	0.12402462
θ_r	0.20900021	**-0.44634144**	-0.2666961	0.40278132

Table E.4: Correlation analysis using the non-recurrent state encoder on the data shown in Figure E.1

Tables E.3 and E.4 show the correlation between the principal components of the learned state representation and the ground truth state when applied on the test data for the recurrent and non-recurrent state representation networks respectively. The most notable difference between the two is that x_r has very little correlation with the learned state representation when using the recurrent encoder network, while the non-recurrent encoder has similar correlation between the three parts of the ground truth state.

E.3 Q-value analysis

The Q-value can be plotted against the position as a measure of the perceived value of a state at that position. Figure E.3a shows the Q-value plotted for the training process when using a recurrent state representation network. Figure E.3b shows the same plot for the non-recurrent state representation network. The difference between the two plots is clear, the perceived Q-value reaches much higher values near the target when the non-recurrent state encoder is used. This is a result of reaching the target more reliably, due to a better state representation.



(a) Q-value versus position during training when using a recurrent state encoder

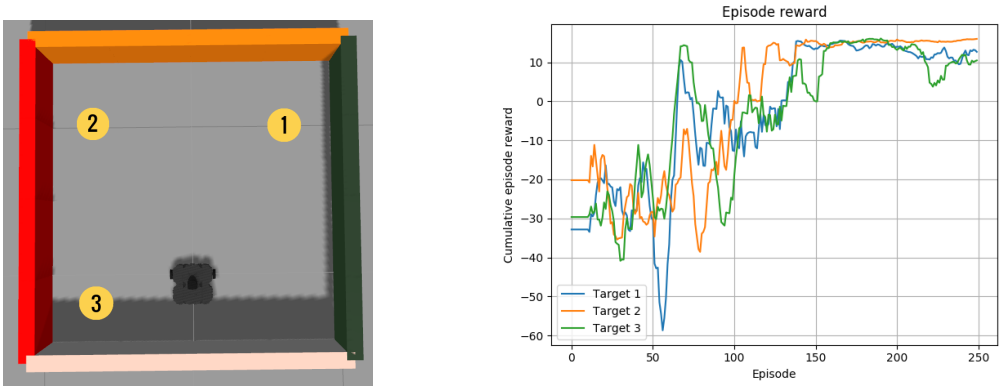
(b) Q-value versus position during training when using a non-recurrent state encoder

Figure E.3: Showing the Q-value versus position during training

F Reaching multiple target locations

A good state representation should enable the reinforcement learning algorithm to learn a good policy for any target. The experiments in this work have only considered the case in which the data on which the SR was trained used the same target as the target location for the policy. However, to make state representation learning more practical and make it have an advantage over end-to-end learning a policy, this SR should generalise to other areas of the environment. To test whether an SR trained on data for one target enables learning to reach other targets in the environment as well, an experiment was performed.

For this experiment, the small square environment was used. The SR was trained using the same dataset on which all state representations of the small square environment were trained, as presented in Figure 5.6a. Figure F.1a shows the target locations which were tested. Target location one is the same location as shown in Figure 4.1a, and also the target location used when gathering the training data. Using this training data the SR encoder network was trained. This encoder network was then used to train a policy for each of the targets. Figure F.1b shows the episode rewards for each of the three targets. This shows that there was no problem generalising to the other target locations, with the time of convergence and the final reward per episode being similar across the different targets.



(a) The tested target locations in the small square environment

(b) Episodic reward achieved during training for the three target locations

Figure F.1: Testing the generalisation of the learned SR to other targets

Bibliography

[1] Baxter (robot), Apr 2020.

[2] Deep deterministic policy gradient¶, 2020. Accessed: 2020-06-10.

[3] Dynamixel sdk, 2020. Accessed: 2020-06-10.

[4] E-manuel turtlebot robot, 2020. Accessed: 2020-06-10.

[5] Powering the world's robots, 2020. Accessed: 2020-06-10.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Pulkit Agrawal, Ashvin Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. Learning to poke by poking: Experiential learning of intuitive physics. *CoRR*, abs/1606.07419, 2016.

[8] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. *CoRR*, abs/1902.01724, 2019.

[9] John-Alexander M. Assael, Niklas Wahlström, Thomas B. Schön, and Marc Peter Deisenroth. Data-efficient learning of feedback policies from image pixels using deep dynamical models. *CoRR*, abs/1510.02173, 2015.

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

[11] Nicolò Botteghi, Ruben Obbink, Daan Geijs, Mannes Poel, Beril Sirmacek, Christoph Brune, Abeje Mersha, and Stefano Stramigioli. Low dimensional state representation learning with reward-shaped priors, 2020.

[12] William Curran, Tim Brys, Matthew E. Taylor, and William D. Smart. Using PCA to efficiently represent state spaces. *CoRR*, abs/1505.00322, 2015.

[13] 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.

[14] I K Fodor. A survey of dimension reduction techniques. Technical report, Lawrence Livermore National Lab., CA (US), 2002.

[15] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[16] Ross Goroshin, Michaël Mathieu, and Yann LeCun. Learning to linearize under uncertainty. *CoRR*, abs/1506.03011, 2015.

[17] Faiza Gul, Wan Rahiman, and Syed Sahal Nazli Alhady. A comprehensive study for robot navigation techniques. *Cogent Engineering*, 6(1):1632046, 2019.

[18] Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control. *Mach. Learn.*, 84(1-2):137–169, July 2011.

[19] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[20] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[21] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.

[22] Herke Hoof, Nutan Chen, Maximilian Karl, Patrick van der Smagt, and Jan Peters. Stable reinforcement learning with autoencoders for tactile and visual data. 10 2016.

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[24] Rico Jonschkowski and Oliver Brock. Learning state representations with robotic priors. *Auton. Robots*, 39(3):407–428, October 2015.

[25] Rico Jonschkowski, Roland Hafner, Jonathan Scholz, and Martin A. Riedmiller. Pves: Position-velocity encoders for unsupervised learning of structured state representations. *CoRR*, abs/1705.09805, 2017.

[26] Maximilian Karl, Maximilian Soelch, Justin Bayer, and Patrick van der Smagt. Deep variational bayes filters: Unsupervised learning of state space models from raw data. 04 2017.

[27] Ronald Kemker, Angelina Abitino, Marc McClure, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. *CoRR*, abs/1708.02072, 2017.

[28] Jens Kober, J. Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013.

[29] Vijaymohan Konda. *Actor-critic Algorithms*. PhD thesis, Cambridge, MA, USA, 2002. AAI0804543.

[30] Timothée Lesort, Mathieu Seurin, Xinrui Li, Natalia Díaz Rodríguez, and David Filliat. Unsupervised state representation learning with robotic priors: a robustness benchmark. *CoRR*, abs/1709.05185, 2017.

[31] Timothée Lesort, Mathieu Seurin, Xinrui Li, Natalia Díaz Rodríguez, and David Filliat. Unsupervised state representation learning with robotic priors: a robustness benchmark. *CoRR*, abs/1709.05185, 2017.

[32] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

[33] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[36] Marco Morik, Divyam Rastogi, and Oliver Brock. State representation learning with robotic priors for partially observable environments data. 2019.

[37] OpenAI. A toolkit for developing and comparing reinforcement learning algorithms.

[38] OpenAI. A toolkit for developing and comparing reinforcement learning algorithms.

[39] Openai. openai/baselines, 2020. Accessed: 2020-06-10.

[40] OpenAI. A toolkit for developing and comparing reinforcement learning algorithms, 2020. Accessed: 2020-06-4.

[41] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.

[42] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. 2019.

[43] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning, September 2017.

[44] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[45] Jahanzaib Shabbir and Tarique Anwer. A survey of deep learning techniques for mobile robot applications. *CoRR*, abs/1803.07608, 2018.

[46] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[47] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[48] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org, 2014.

[49] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[50] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930.

[51] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[52] Manuel Watter, Jost Tobias Springenberg, Joschka Boedecker, and Martin A. Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. *CoRR*, abs/1506.07365, 2015.

[53] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.