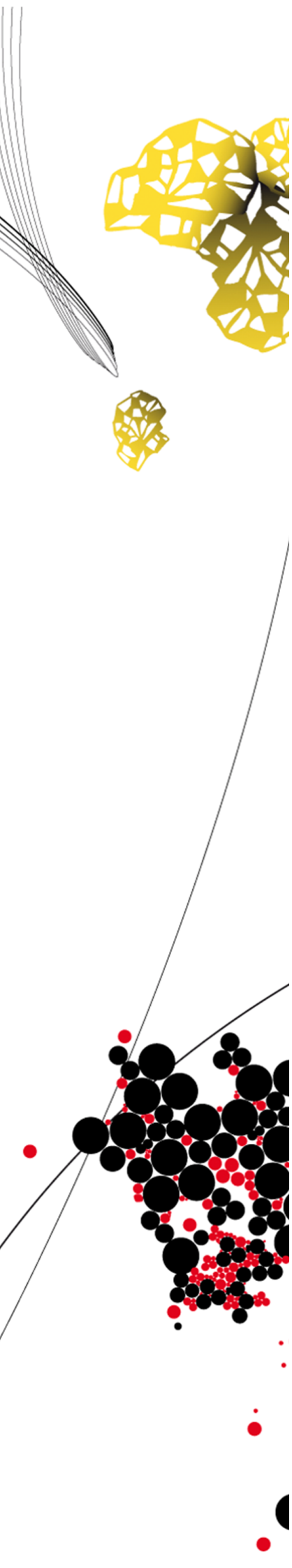


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Robot Navigation in Dynamic Evironment Based on Reinforcement Learning

Cijun Ouyang

MSc. Final Thesis
December 2020

Supervisors:
Dr. M.Poel
N.Botteghi MSc
Prof.Dr.Ir. G.J.M. Krijnen

Data Management & Biometrics
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Acknowledgement

My final thesis is coming to an end after eight months of work. Here I would like to express my sincere appreciation to all those who helped me and supported me during this really important and precious period.

Firstly, I would like to thank my supervisors for their help and encouragement. Special gratitude goes to my daily supervisor MSc Nicolò Botteghi for him sharing his experience and knowledge in the field of reinforcement learning and bringing ideas of new methods and solutions for my work. In addition to that, his detailed guidance and great patience on my daily work of thesis and code contributed a lot to the enthusiasm that I have maintained during my research. Also, a special thank goes to dr. Mannes Poel for his support of my choice in the direction of my thesis at the beginning of the time. Next to that, the novel advice and constructive comments on my research during every meeting encouraged me a lot to explore the possibilities of my project and improve my research ability. I also thank my remaining committee Prof.dr.ir Gijs Krijnen for becoming my committee members.

Secondly, I would thank my parents to make a chance for me with this precious experience at UT. Without their support and love, I would not have become the person I am now. I also can't forget the encouragement from all my friends. They gave me really important support in my study and company in my daily life during these two years in the Netherlands, which inspired my life.

Last but not least, I would like thank to my violin and the Musica Silvestra Orkestra. I'm grateful for all the unforgettable rehearsals and concerts I had with our amazing conductor Peter and excellent concertmaster Maxim. I can't forget the fun and enthusiasm of the fantastic orchestra and symphony music gave me during these two years, which relaxed me a lot from my study and gave me infinite energy.

Thanks to everyone.

Abstract

Mobile robot navigation attracts research and industrial interests in various fields in recent years. Autonomous navigation in an environment has always been a challenge for mobile robots, especially in a dynamic environment. There are various approaches to solve this problem for mobile robots, but most of the approaches need a model of the entire map and precise prior knowledge of the environment, which is difficult to implement in the real world. Therefore, motivation is formed to apply reinforcement learning(RL) for this navigation task in an unknown environment. Since an optimal route to reach the target can be explored by RL through trial-and-error interactions with the unknown environment and gaining the maximum reward, which doesn't need any prior knowledge. This project aims to implement mobile robot navigation in an unknown dynamic environment with the reinforcement learning method. Deep Q-network(DQN) is used in this project because of the advantage of the training stability.

To obtain an optimal policy for path planning with high efficiency and shorter trajectory, we explore several kinds of reward functions and design a proper one for the task in dynamic environments concerning the features of the current states receiving from the environment. Laser sensors are used to obtain the distance information of the target and obstacles around the robot. The two main metrics, Q-value loss and accumulated reward are considered to evaluate the performance of the reward functions. Then it is validated that the reward function concerning both distance and orientation information performs best among the proposed reward functions with low loss value, high accumulated reward, and high stability.

Another problem to solve in this project is to extract high-dimensional observation from the environment and compress the observation to low-dimensional states. We use an RGB camera to get observation images and use an auto-encoder to implement state representation of the images. We proposed two methods to extract main features from the dynamic environment. The first one is combining the encoded states from the auto-encoder with laser measurements and additional position states to get precise positions of moving obstacles. The second is inputting a sequence of observations to auto-encoder to get the motion pattern of the moving objects. It is proved that with these methods, the positions of the moving obstacles

can be tracked, which improves the success rate of the navigation significantly.

Contents

List of Figures

List of Acronyms

RL	Reinforcement Learning
DQL	Deep Q-Learning
DQN	Deep Q-Network
DDQN	Double Q-Network
SRL	State Representation Learning
MDP	Markov Decision Process
ReLU	Rectified Linear Unit
CNN	Convolutional Neural Network

Chapter 1

Introduction

1.1 Motivation

With the continuous development of robot technology, intelligent mobile robots have been applied in various fields and play an increasingly important role in home daily services, medical services, industry, agriculture, military, and other fields. Autonomous navigation, which meets the fundamental needs of mobility for real-life robots, plays an important role in the fields of autonomous and intelligent mobile robots. As a basic research content, robot navigation in an unknown environment is still a challenge, which makes it a technology focus. One of the advantageous approaches for navigation in an unknown environment is reinforcement learning(RL).

RL enables an agent to autonomously discover an optimal behavior through trial-and-error interactions with its environment [3]. The agent decides the next action based on the information perceived from the environment. Then it receives feedback from the environment in the next state to tell whether the taken action is good or bad(i.e. the reward). By interacting with the environment, the agent would learn to discover a path to reach the target. The goal of RL is to maximize the accumulated long-term reward. Encouraged by this goal, the agent could explore the optimal policy mapping from state to action and discover the shortest trajectory.

The problems that could be encountered when applying reinforcement learning in robot navigation. Since getting the reward from the environment is the only way for the agent to evaluate its learning performance and results in the decision on the next action, designing a proper reward function is important and challenging. Inappropriate reward function, such as reward sparsity case, would lead to algorithm divergence [2] and make the agent fail to achieve the goal. In the robot navigation case, the design of the reward function considers the features of the current state responded from the environment. Next to that, the discretization of continuous states and actions of the robots from observation always results in an exponential explosion of states and lead to an expensive computation and low convergence rate [4]. The

approaches for observation compression are discussed in Section 2.2.1 and auto-encoder is proposed to be used in this project to reduce the dimension of the state space.

This project focuses on the navigation problem for robots in unknown dynamic environments based on RL. In dynamic environments, moving targets and obstacles are considered. Deep Q-network(DQN) is the mainly discussed approach to implement the goal, which has good training stability compared with traditional Q-network [5].

Secondly, we designed different reward functions based on the responses from the dynamic environment to improve learning efficiency. The performance of these reward functions is evaluated. After a proper reward function being defined, we use an RGB camera to obtain the information from the environment to get a visual observation. An auto-encoder is applied to compress the observation to state representation of the camera image for an efficient learning process. We explore several methods to extract meaningful information from the environment and encode important features from the observation to help the agent making a better decision on the next action. Then we compare the performance between the methods with these different observation data.

1.2 Research Questions

Based on the motivation proposed in section 1.1, the following research questions are formulated:

1. RQ1: What is a proper reward function for the navigation task in a dynamic environment to help the agent learning the policy and accelerate the learning efficiency?

2. RQ2: In the navigation problem, how to reduce the dimension of the state space and extract meaningful information with auto-encoder in a dynamic environment?

1.3 Report Organization

In Chapter 2, we introduce the concept of some basic methods and algorithms to solve the navigation problems. This part also includes the introduction of relevant methods used by previous works, together with their limitation and disadvantages. Chapter 3 discusses solutions based on the concept introduced in chapter 2. Deep reinforcement learning algorithms are discussed to provide a particular method for

the final project. Especially, a novel path planning algorithm by means of local path planning with double Q-network is briefly analyzed. To answer the research questions formulated in Section 1.2, we propose several approaches in Chapter 4. Firstly we introduce the architecture of the neural network we use in DQN. Then we present some different kinds of reward functions we use in the experiments and what are the differences between these reward functions. Secondly, we introduce the framework that applying state representation learning in the training of RL to compress the observation state space. In Chapter 5, we first describe the environments and hyperparameters we use in the experiments. Then we present the setup of the experiments corresponds to each research question. Next to that, we show the results of the experiments in Chapter 6 and analyze the possible reasons for the results to answer the research questions in Chapter 7. Finally, in Chapter 8 we draw the conclusions of the experiments and put forward the further researches that we will discuss in the future.

Chapter 2

Background

This chapter introduces the basic concepts that we use to implement robot navigation. Section 2.1 gives an overview of RL, together with its key element Markov Decision Process(MDP). Then deep Q-learning(DQL) is presented, which is the method mainly used in this project.

Next, the concept of state representation learning(SRL) is introduced in Section 2.2. This is used for feature learning dealing with high-dimensional problems. The methods used in previous work to solve the problem of high-dimensional curse with SRL are discussed in Section 2.2.1 and one of the approaches are introduced in Section 2.2.2.

2.1 Reinforcement Learning

Reinforcement learning(RL) is a goal-directed learning method to learn how to maximize the value of the reward in the predefined task, which is actually a learning process of mapping state to action. Compared with the classic supervised learning and unsupervised learning problems of machine learning, the main feature of reinforcement learning is learning from interaction. During the interaction with the environment, the agent continuously learns knowledge based on the obtained rewards or punishments and adapts to the environment. The main difference between RL and supervised learning is that there is no output value of training data prepared for supervised learning. Reinforcement learning only has a reward value obtained at the next time step, which is not the same as the output value of supervised learning. At the same time, each step of RL is closely related to the time sequence. The paradigm of RL is very similar to the process of humans learning knowledge, and it is for this reason that RL is regarded as an important way to achieve general AI.

The classic quadruple in RL [3] is denoted as $< A, S, R, P >$. A represents the set of actions of the agent. The action a_t taken by the agent at time t is a

certain action in its action set. S is the state set of the environment that the agent can perceive. The state s_t of the environment at time t is a state in its state set. R is a reward function, representing the reward or punishment of the agent. The reward r_{t+1} corresponding to the action a_t taken at state s_t at time t is obtained at time $t+1$. The reward function, which is the goal of RL, can be viewed as a mapping from observed environment variables to the reward value, measuring the satisfaction of that state or the taken action. The goal of a reinforcement learning agent is to maximize a cumulative reward for long-term actions. The reward function determines whether the current decision of the action is a good decision for the agent. P is the state transition probability function, which represents the probability that the environment is transferred to the s_{t+1} state after an action a_t is performed in s state.

The core problem in RL is to learn a good policy for sequential decision problems by optimizing a cumulative reward signal. Policy, defined as π, is the choice of action a made by an agent to make in the state s. A policy can be regarded as mapping to action a after the agent explores the environment. If the strategy is stochastic, the policy selects the action based on the probability $\pi(a \mid s)$ of each action; if the strategy is deterministic, the policy directly selects the action $a = \pi(s)$ based on the state s.

2.1.1 Markov Decision Process and Value Function

As it mentioned before, the next state of the environment s_{t+1} depends not only on the current state s_t but also on the action a_t taken by the agent at time t. The interaction model of the transition can be modelled as MDP [6], which is defined as:

$$P(S_{t+1}, R_{t+1} \mid S_0, A_0, R_1, ..., S_t, A_t) = P(S_{t+1}, R_{t+1} \mid S_t, A_t) \tag{2.1}$$

where R is the reward, S is the state and A is the action. According to the property of the MDP, the response of the environment at the next time $t+1$ is only related to the information from the current state s_t and action a_t. The interaction model between agent and environment is shown in Figure 2.1.

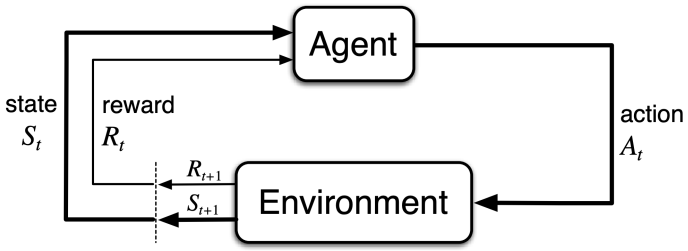


Figure 2.1: The agent-environment interaction model

The value function $V_\pi(s)$, defined as the average reward of action in the long-term, can be estimated based on MDP, which is depend only on the current state s:

$$V_\pi(s) = \mathbb{E}[G_t \mid S_t = s] \tag{2.2}$$

where G_t is the return of state s_t at time t:

$$G_t = R_{t+1} + \lambda R_{t+2} + ... = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1} \tag{2.3}$$

$\lambda < 1$ is the discount factor which means the current return is relatively important and the impact reduces as time passing.

Also the action-value function $Q_\pi(s,a)$ for policy π, which take action into account, is defined as:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi[R_{t+1} + \lambda R_{t+2} + \lambda^2 R_{t+3}... \mid S_t = s, A_t = a] \tag{2.4}$$

To solve the problem of RL means to find an optimal policy π for an agent in the process of interaction with the environment, which gains more reward than other policies all the time. The optimal policy is expressed as $\pi*$. Generally speaking, it is difficult to find an optimal policy, but a better policy will be determined by comparing the advantages and disadvantages of several different policies, that is, the local optimal solution. The search for a locally optimal policy will be achieved by searching for a better value function. The optimal state value function is defined as the maximum value of state value functions generated under all policies:

$$V*(s) = \max_\pi V_\pi(s). \tag{2.5}$$

And the action-value function is also defined in the same way:

$$Q*(s,a) = \max_\pi Q_\pi(s,a). \tag{2.6}$$

As long as the largest state value function or action-value function is found out, then

the corresponding policy $\pi*$ is the solution to the RL problem.

2.1.2 Deep Q-Learning

Q-learning is a classical algorithm of RL, which is a table method based on the past state, policies, and iteration Q value. Q means action-utility function, to evaluate the action decided in a specific environment is good or bad. In a simple case, the combined amount of states and actions is finite. So after training the Q values will be filled in a table corresponding to different state s and actions. The learned policy π will choose the action with maximal Q value. The Q value is updated following:

$$Q(s,a) \leftarrow (1-a)Q(s,a) + \alpha[r(s,a) + \gamma max_a Q(s',a)], \tag{2.7}$$

where α is the learning rate; γ is the discount factor; r is the current reward.

The problems of Q-learning are, on the one hand, the available state and action space of Q-learning is very small because of the limited size of the Q table, which is impractical in complex cases with large state and action space; on the other hand, Q-learning could not handle a new state never appeared before. In other words, Q-learning has no predictive ability, that is, no generalization ability [7].

To solve the problem, the optimal action-value function will be parameterized by:

$$Q(s,q;\theta) \approx Q^*(s,a) \tag{2.8}$$

where θ is the Q-network parameter for neural network weights. Reinforcement learning has instability when a nonlinear function approximator, such as a neural network is used to represent the action-value function and will result in divergence [8].This instability has resulted from the correlations present in the sequence of observations and the correlations between the action values (Q) and the target values.

DeepMind proposed a mechanism called experience replay [9] to deal with the instability problem, which stores the agent's experiences, environment state, action, and reward at each time-step in a memory buffer. The mechanism reduced correlations in the observation sequence by sampling random data from the pool of the dataset when training the network.

Then the method of deep Q-network(DQN) is improved in 2015 by added a target Q-network, which updates parameters with a low rate after comparison with predicted Q-network [10]. So the target Q-network and predicted Q-networks are built with the same structure and use different parameters. The parameters in the target Q-network is denoted as $\theta_i{}'$ and parameters in the predicted Q-network is θ_i. The values in the target Q-network are only updated with the predicted Q-network

in every C steps, thus the update of the target Q-network is delayed and over-fitting is avoided.

The update mode of Q value is similar to that of Q-learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'}\hat{Q}(s',a') - Q(s,a)]. \tag{2.9}$$

The improved loss function is:

$$L_i(\theta_i) = E\left[(r + \gamma max_{a'}\hat{Q}(s',a';\theta_i{}') - Q(s,a;\theta_i))^2\right], \tag{2.10}$$

where $\hat{Q}(s',a';\theta_i{}')$ is target Q-network and $Q(s,a;\theta_i)$ is predicted Q-network(same meaning of the notation in the following content).

DQN applied two key technologies:

1. Experience Reply: put the collected samples into the sample pool first, and then randomly select a sample from the sample pool for network training. This method removes the correlation between samples and makes them independent from each other.

2. Fixed Q-target network: the existing Q value is needed to calculate the network target value, which is provided by a network with a slow updating rate. This improves the stability and convergence of training.

The DQN algorithm is described in Algorithm 1.

Algorithm 1 Deep Q-learning with experience replay [11])

Require: Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for each iteration **do**
Initialize S
for each environment step **do**
Observe state s_t
With probability ϵ select a random action a_t, otherwise select $a_t = argmax_a Q(s,a;\theta)$
Execute a_t and observe next state s_{t+1} and reward $r_t = R(a_t,s_t)$, Set $s_{t+1} = s_t$
Store transition(s_t,a_t,r_t,s_{t+1}) in D
Sample random minibatch of transitions (s_t,a_t,r_t,s_{t+1}) from D
Set $\begin{cases} r_j, & if\ episode\ terminates\ at\ step\ j+1 \\ r_j + \gamma max_{a'}\hat{Q}(s',a';\theta), & otherwise \end{cases}$
Perform a gradient descent step on $(y_i - Q(s,a;\theta))^2$ with respect to the network parameters θ
Every C steps reset $\hat{Q}$=Q
End **for**
End **for**

2.2 State Representation Learning

In the robotics problems, the control of robots is based on the sensor data getting from the environment. When the high dimensional data provided by sensors(e.g.camera), the robot objective can always be expressed in a low-dimensional space as the state of the system, which filtered much inessential original data and only keeps substantial information. Instead of directly using original data, learning from low dimensional representations, the tasks would be solved more efficiently [12]. State representation learning(SRL) is a particular case for feature learning with low dimension. The objective of SRL is to transform original observations into a state set remaining the most representative features for policy learning, which is based on time steps, actions, and optionally rewards [1].

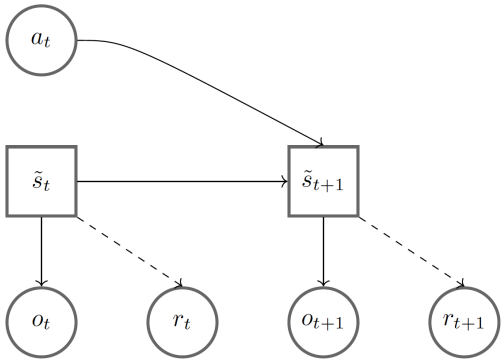


Figure 2.2: General model : circle are observable and square are the latent state variables [1]

The SRL formalism is based on the one for reinforcement learning introduced in [3]. The general model of SRL is illustrated in Figure 2.2. The environment is defined as E and actions taken at time step *t* are defined as $a_t \in A$, where A is the action space. The arrow from $\tilde{s}_t$ to $\tilde{s}_{t_1}$ in Figure 2.2 reflects the transition of the true state after the agent takes the action. The true state is denoted as $\tilde{\mathcal{S}}$. $o_t \in \mathcal{O}$ represents the observation of the environment $\mathcal{E}$ that the agent receives from sensors, where $\mathcal{Q}$ is the observation space. Optionally, the reward given at $\tilde{s_t}$ to the agent is denoted as r_t and this is present as one of the goal of SRL. SRL aims to learn a mapping ϕ of the observation to the current state $s_t = \phi(o_t)$, where $\phi(o_t)$ also includes action a_t and rewards r_t as parameters.

Generally applied approaches of SRL such as auto-encoder, forward model and inverse model are introduced in following([1]) based on the notations introduced above. Reconstruction the observation is one of the SRL strategies for learning the mapping function ϕ with a encoder and minimizing the reconstruction error between the original observation and the reconstructed observation with a decoder ϕ^{-1}. The

state s_t and reconstructed observation are written as:

$$s_t = \phi(o_t; \theta_\phi) \tag{2.11}$$

$$\hat{o}_t = \phi^{-1}(s_t; \theta_{\phi^{-1}}) \tag{2.12}$$

Here $\hat{o}_t$ is the reconstructed observation. θ_ϕ and $\theta_{\phi^{-1}}$ are the parameters of encoder and decoder respectively. As shown in Figure 2.3, the difference between o_t and $\hat{o}_t$ is calculated as the reconstruction error.

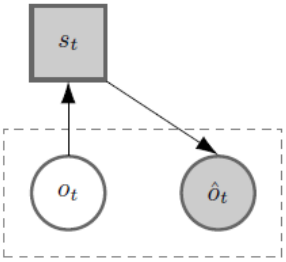


Figure 2.3: Auto-Encoder: observation reconstruction [1]

Learning a forward model, shown in Figure 2.4, also helps to learn state representations by encoding the substantial information to predict the next state. The next state s_{t+1} is predicted from s_t and a_t or o_t. In order to learn the mapping ϕ from observation o_t to the state s_t, the prediction process is encoding from o_t to s_t and then transition from s_t and action a_t to the predicted state $\hat{s}_{t+1}$. Then the error between the predicted state $\hat{s}_{t+1}$ and the actual next state s_{t+1} at $t+1$ are computed to learn for the model. After that the error is back-propagated from state $\hat{s}_{t+1}$ to state s_t and back to observation o_t.

$$\hat{s}_{t+1} = f(s_t, a_t; \theta_{fwd}) \tag{2.13}$$

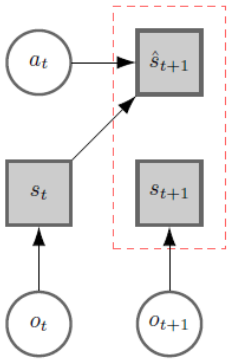


Figure 2.4: Forward Model: predicting the next state [1]

To predict action a_t from states s_t and s_{t+1} or observations o_t and o_{t+1}, an inverse model is applied in SRL. To learn the mapping ϕ from observation o_t to state s_t, the framework firstly projecting o_t and o_{t_1} onto s_t and s_{t+1} respectively. Then it predicts the action $\hat{a}_t$ based on the transition from s_t to s_{t+1}. The error is calculated between the predicted action $\hat{a}_t$ and the true action a_t for the prediction and then back-propagated to the encoding model.

$$\hat{a}_t = g(s_t, s_{t+1}; \theta_{inv}) \tag{2.14}$$

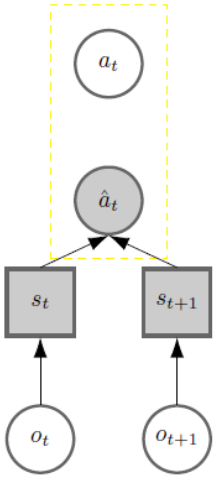


Figure 2.5: Inverse Model: predicting the action [1]

2.2.1 Observation Compression for Robot Navigation

In the robot navigation cases, the more complex the environment is, the higher dimensionality of the observation and action is generated. So it will be hard to learn the state representation, as only the substantial information and smaller dimension of the state space we needed. To solve this problem, solutions for compressing state and action spaces are discussed in previous work:

Methods used in [13] [7] applied auto-encoder to reduce state space, which will be discussed in details in Section 2.2.2. The method compressed the state representation using neural networks to extract the important features from original input data through the hidden layer of the auto-encoder. Kimura et al. [14] proposed a method to combine the auto-encoder with Q-network: training a network by an auto-encoder; deleting the decoder layers, and adds a fully-connected layer on the top of encoder layers as an input to DQN; training policies by the DQN algorithm initialized by the pre-trained network parameters.

The curiosity-driven method proposed in [15] created an inverse model to extract

features that impacted by action and used a forward model to predict the next state based on the extracted features. This feature space can be learned by training a deep neural network with two sub-modules: the first one encodes the raw state (s_t) into a feature vector $\Phi(s_t)$; the second one takes the feature encoding $\Phi(s_t)$, $\Phi(s_{t+1})$ of two consequent states as inputs and predicts the action (a_t) taken by the agent to move from state s_t to s_{t+1}.

The Value Prediction Network(VPN) applied by Oh et al. in [16] combined model-based RL and model-free RL in a unified framework. The encoding module maps the observation to the abstract state using neural networks(e.g., CNN for visual observations). Thus, an abstract-state representation will be learned by the network.

2.2.2 Auto-Encoder

Auto-encoder is widely used to learn state representation for data compression implemented by a multi-layer neural network. It uses the backpropagation algorithm to make the target value equals the input value. Given a data sample, auto-encoder aims at extracting features and generating low-dimensional data as a self-supervised model. In the robot navigation case, as mentioned in Section 2.2.1, the auto-encoder is applied to compress the state space got from observation and build a simpler representation of the states to improve the training efficiency.

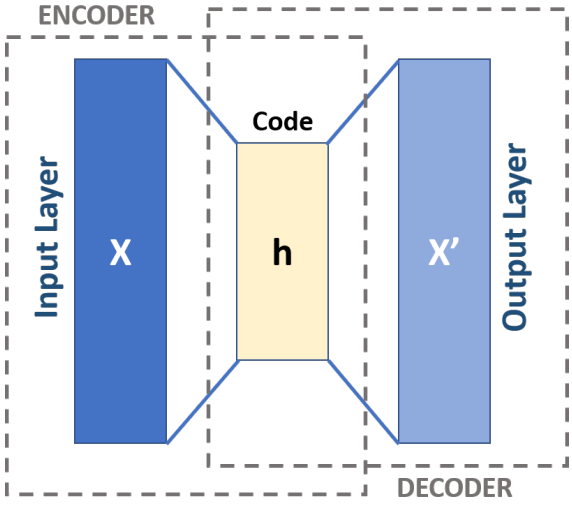


Figure 2.6: The structure of auto-encoder

Auto-encoder has three components as shown in Figure 2.6: input layer, hidden layer created by encoder block, output layer created by decoder block. The hidden layer contains lower-dimensional representing features of input data and output layer reconstructs data from the vector from hidden layer. The encoder function and

decoder function are defined as ϕ and ψ respectively:

$$\begin{aligned} &\phi : \mathcal{X} \to \mathcal{F}, \\ &\psi : \mathcal{F} \to \mathcal{X}, \\ &\phi, \psi = argmin_{\phi,\psi} \left\| X - (\phi o \psi) X \right\|^2, \end{aligned} \tag{2.15}$$

where $\mathbf{x} \in R^d = X$ is the given inputs.

In the simplest case with only one hidden layer, the encoder function maps the input $\mathcal{X}$ to the latent space $\mathbf{h} \in R^p = \mathcal{F}$:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{2.16}$$

Here σ is activation function that always uses non-linear function such as ReLU(Rectified Linear Unit) and sigmoid. $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector, which are updated every iteration through back-propagation during training.

Similarly, the decoder function maps the latent space $\mathcal{F}$ to the output as a reconstruction $\mathbf{x}'$, which has the same shape as the input $\mathbf{x}$:

$$\mathbf{x}' = \sigma' \left(\mathbf{W}'\mathbf{h} + \mathbf{b}' \right). \tag{2.17}$$

To minimise the errors between the original input data and the reconstruction data, loss function is used to training the neural network in back-propagation process:

$$\mathcal{L}\left(\mathbf{x}, \mathbf{x}'\right) = \left\| \mathbf{x} - \mathbf{x}' \right\|^2 = \left\| \mathbf{x} - \sigma' \left(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}' \right) \right\|^2. \tag{2.18}$$

After being compressed and encoded, the high-dimensional original data is represented by a low-dimensional vector, The latent space $\mathcal{F}$ contains the typical features of the original data and has lower dimensionality than the input space X. So the encoder block is useful for data compression.

Chapter 3

Robot Navigation Based on Deep Reinforcement Learning

This chapter first discusses the methods applied in robot navigation based on Deep Reinforcement Learning. Then Section 3.2 introduced a specific approach proposed by a previous paper [2], which applied Double Q-Network(DDQN) to dynamic path planning in an unknown environment.

3.1 Robot Navigation

The methods proposed in previous work [17] are categorized into two types, based on the global environment and local environment information. In the first category, the methods use a priori information to reconstruct the environment and try to find the most optimal path [18] [19]. On the other side, the local navigation method uses local environment data detected by sensors to plan actions for robots. This kind of method is able to achieve real-time path planning [20] and is applied in dynamic environment. Local learning-based navigation algorithms include Deep Learning (DL) and Deep Reinforcement Learning (DRL) [21]. In this paper, we mainly discuss methods based on Deep Reinforcement Learning.

DRL learns navigation policies by observation collected from interactions between the agent and the environment. For example, Oh et al. [16] proposed a value prediction network(VPN) which combines model-based RL and model-free RL in a unified framework. It learned to predict values via Q-learning and rewards via supervised learning, then implemented exploration in a 2D stochastic static environment. Zhelo et al. [22] proposed curiosity-driven exploration strategies to argue robots' abilities to explore complex unknown environments. However, these works mainly focus on navigation in static environments, which are unpractical for most realistic applications since various kinds of moving objects are required to be taken into

consideration.

To navigate in a dynamic environment, Lei et al. [2] applied DDQN algorithm to enhances the ability of the dynamic obstacle avoidance and local planning of the agents in the environment. Bae et al. [23] combined the RL algorithm with the path planning algorithm of the mobile robot to compensate for the demerits of conventional methods by learning the situation where each robot has mutual influence. Zeng et al. [21] proposed a Memory and Knowledge-based Asynchronous Advantage Actor-Critic approach, which improved A3C algorithm by using memory mechanism, domain knowledge, and transfer learning. Yen et al. [24] proposed three methods: forgetting Q-learning, feature-based Q-learning, and hierarchical Q-learning. They used forgetting Q-learning to improve performance in a dynamic environment by reserving the navigation paths. Feature based RL is accomplished by using a feature identification method to process state inputs. Hierarchical Q-learning is proposed to lead towards both goals of navigation in a dynamic environment and knowledge transference among multiple agents.

In the navigation tasks, the reward sparsity problem will result in slow convergence or divergence of the algorithm. Several methods are put forward to solve the reward sparsity problem. Deepak Pathak et al. [15] applied Intrinsic Curiosity Module(ICM) to the navigation. This module makes the agent explore the environment better through the additional reward generated by ICM. In sparse reward tasks, it is low efficient for the agent to explore the environment. Then ICM is designed to evaluate the familiarity of the environment. This is based on the current state and the action to predict the next state, and then calculate the deviation from the actual state. This familiarity is the extra reward, which is called curiosity. Mirowski et al. [25] implemented navigation search is based on A3C with visual information. It adds two auxiliary training tasks in addition to the convolutional neural network(CNN) with a stacked LSTM layer. One is the depth prediction of each step that aimed to encourage learning of obstacle avoidance. Another one is loop closure prediction to detect whether the local position is already visited in a local trajectory. The determination of whether the robot has passed through the same location is used to train the CNN network based on the data in the simulation environment. Further, an output is introduced into the well-trained CNN network to output the position information, and the position of the robot will be quite accurate after training for a period of time.

3.2 Path Planning with Double Q-Learning Network

As mentioned in Section 3.1, Lei et al. [2] proposed a method based on DDQN (introduced in Appendix A in details) for path planning in dynamic environment with random target and moving obstacles. Based on DQN, DDQN produces more ac-

curate value estimates [5] [26], which reduces over-estimations by evaluating the greedy policy with online network and estimate values with the target network. Lei et al. achieved local path planning with DDQN using Lidar sensors and designed reward function according to distance from the target point.

3.2.1 Path planing with DDQN

In the local path planning implemented by Lei et al. [2], the input data is the information getting from lidar sensors, which consists of angles in 360 degrees and distances to obstacles within a circle, and the output is the chosen action.

Considering the input of the network is large and will result in expensive computation cost, a convolutional neural network(CNN) is used in DDQN to extract features and reduce the dimensions of input. As the lidar data reflects the position of obstacles and the range of free zone, CNN can reduce the network parameters efficiently in a dynamic environment.

The reward function is designed to be related to the positions of target point and obstacles. If the position of the agent is equal to the target's, which means the agent reaches the target, it will gain a reward of 1. If the position of the agent is equal to the obstacle's, which means the agent crashes to an obstacle, it will get a penalty of -1. In other cases, the agent gets a penalty of -0.01. The agent will plan a path avoiding the obstacles and reaching the target by trial-and-error learning. The reward function is designed as:

$$r = \begin{cases} 1(p(x',y') = g(x,y)) \\ -0.01 \begin{pmatrix} (p(x',y') \neq g(x,y)) \\ (p(x',y') \neq o(x,y)) \end{pmatrix} \\ -1(p(x',y') = o(x,y)) \end{cases} \tag{3.1}$$

where $p(x',y')$ is the current position, $g(x,y)$ and $o(x,y)$ present target point and obstacle position respectively.

To improve the sample space and avoid the main states distributed in free space, the radius L from the initial position to the target point is designed to be increased gradually from a small value. The probability that the agent reaching the targets will increase at the beginning stage and a positive incentive in the sample space is ensured because the agent is close to the target point. The radius L increases with

the neural network being updated gradually.

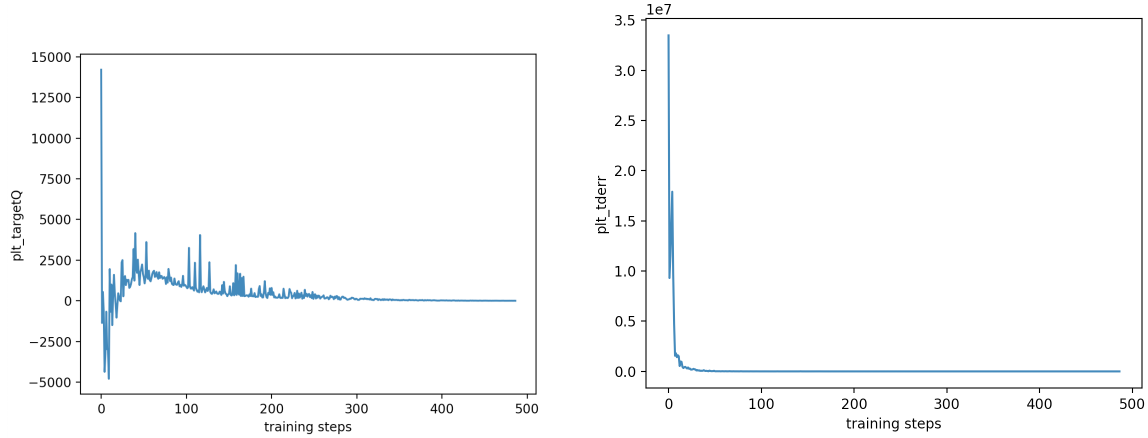
$$L = \begin{cases} L_{min} & (n \leq N_1) \\ L_{min} + \sqrt{\frac{(n-N_1)}{m}} & (N_1 < n < N_2) \\ L_{max} & (n \geq N_2) \end{cases} \tag{3.2}$$

where L_{min} and L_{max} are the initial and maximum value of radius L respectively; N_1 and N_2 are the thresholds of the iteration steps and they are depended on the training parameters; n is the current iteration step and m is the batch size.

3.2.2 Results of Path Planning Simulation

In the experiments, Lei et al. [2] trained the proposed DDQN network in a dynamic environment built by the Pygame module. The environment includes static walls and dynamic obstacles moving in some fixed areas. To verify the DDQN approach, we reproduce the method introduced in [2] and get the results in Figure 3.1 and Figure 3.2 after training 5000 epochs. Figure 3.1a and 3.1b show the loss function values of both estimation and target Q-network decrease continuously and convergence to less than 1 finally. Figure 3.2 presents the curve of average cumulative reward. The reward value increases gradually finally reached a stable stage. This result means the agents achieved to learn the environment and reach the target point after training. As it is noticed in Figure 3.1a and Figure 3.1b, a high peak of loss value and a low peak of reward value appear at about the first 1000 epochs and then it increases. This is because that when the agent exploring the environment at the initial stage of the training process, new batches of the information are received from the environment and the policy is always updated based on the new information.

The application of the DDQN algorithm in the path planning task obtained a satisfying result, which is capable and flexible in a dynamic unknown environment.



(a) Training curve of the loss function of estimation Q-network

(b) Training curve of the loss function of target Q-network

Figure 3.1: Simulation Results

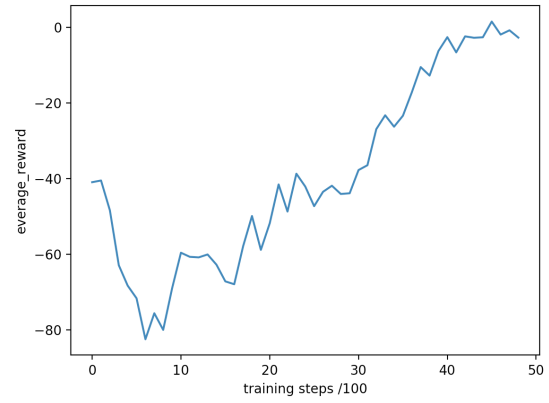


Figure 3.2: Average Cumulative Reward

Chapter 4

Methodology

In this chapter, we present the architecture of the DQN we use in this project and discuss the answers to the research questions based on the current method. Approaches are elaborated to answer the research questions formulated in Section 1.2. The experiments designed according to these approaches are described in the next chapter.

4.1 The Neural Network in DQN

In this project, we achieve our navigation goal with DQN that introduced in Section 2.1.2. The architecture of the neural network in DQN is shown in Figure 4.1.

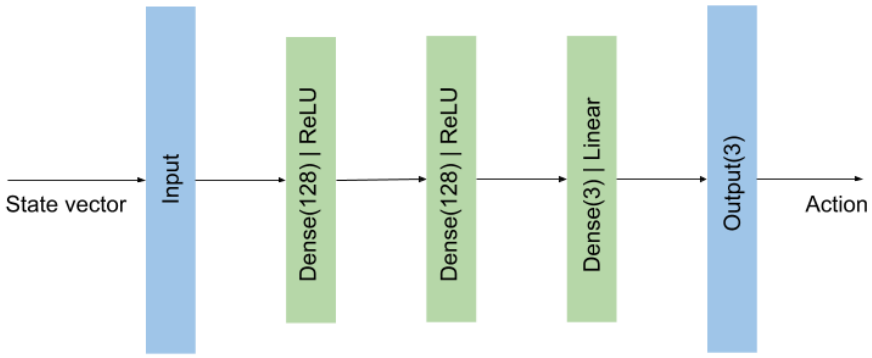


Figure 4.1: The neural network architecture in DQN

The input layer is a one-dimensional state vector. The content of the state vector varies from different experiments for different research goals. In the preliminary stage and reward function design in Section 4.2, We use 16 laser sensors on the robot to detect the information of the environment over 360 degrees. In the observation compression cases in Section 4.3, we use a camera to get RGB and grayscale images as observations and the input to DQN is the encoded latent state space.

Also, some auxiliary states such as the position of the robot, the obstacles and the target will be taken into account to help to learn more information from the environment. The size of the state vector will be mentioned in the description of every experiment in Chapter 5.

Since the input of DQN is a low-dimensional state vector, we only use three full-connected(dense) layers in the neural network. The first two full-connected layers contain 128 neurons in each layer and the rectified linear units(ReLU) is the activation of the dense layers. Then the next layer is a full-connected layer with a size of 3, which is the output layer. The output of the last full-connected layer is the action space. This neural network can extract the features from the input state vector and learn the best route in the dynamic environment efficiently.

4.2 RQ1:Reward Function Design

The goal of RL algorithm is to maximize the long-term accumulated reward so that an optimal policy will be learned for the agent to reach the target. Given the state of the environment and the action taken by the agent, the reward function returns a value reflecting whether the action is good or bad for reaching the target. Thus, reward shaping is an important aspect and it is challenging to design a good reward function. In the navigation problems, the reward function is formulated as a bonus given when the agent reaches the target and penalty given when a collision happens. However, if the state-action space is too large, the possibility of finding a reward will decrease so that it will not learn an optimal policy. Therefore, a more sophisticated reward function is needed in navigation problems, such as take the distance between the agent and the target into account, to make the algorithm learning more efficiently. Several concepts of reward shaping are discussed in this section and these methods will be evaluated in the experiments described in the next chapter.

Exponential Euclidean Distance In this case, the agent will receive a penalty according to the exponent of Euclidean distance between the current position and the target position. The longer distance of the current position to the target point, the fewer reward is given. The reward is formulated as given:

$$r = 1 - e^{\gamma d} \tag{4.1}$$

where d is the Euclidean distance between the agent and the target and γ is a parameter of the decay rate of the exponent that can be tuned.

In addition to the dense reward described above as the distance-based reward, the sparse reward is defined when the agent collides with obstacles or reaches the target: if the agent collides with an obstacle or the wall, the penalty is -20; if the agent reaches the target position, it gets +20. This sparse reward is also defined in the same way in the following reward functions. So the distance-based reward function is designed as:

$$R(s,a,s') = \begin{cases} 20, & \text{if reach the target} \\ 1 - e^{\gamma d}, & \text{otherwise} \\ -20, & \text{if collide the obstacle} \end{cases} \tag{4.2}$$

In environments with dynamic obstacles, an additional reward function based on the distance to the closest obstacle is added. It helps the agent with collision avoidance. When the agent is moving inside a range around the obstacles, which means the minimum distance from the obstacles is less than a pre-defined threshold, the penalty value is computed as inversely proportional to the minimum distance from the obstacles. The closer to the obstacle, the penalty is larger. The complete distance-based reward function in this case is designed as:

$$R(s,a,s') = \begin{cases} 20, & \text{if reach the target} \\ 1 - e^{\gamma d_1} + (-A)/d_2, & \text{if } d_2 \leq \text{threshold} \\ 1 - e^{\gamma d_1}, & \text{otherwise} \\ -20, & \text{if collide the obstacle} \end{cases} \tag{4.3}$$

A is the scaling factor. d_1 is the distance to the target. d_2 refers to the minimum distance to the obstacles. The threshold is set that if the distance between the agent and the nearest obstacle is less than it, the penalty related to the distance to the obstacle is given(same in the following reward functions).

Orientation-based Reward As an auxiliary reward to the distance-based reward, the orientation of the agent is taken into account. This kind of reward is used to avoid the agent stay in a position escaping the penalty and find an optimal route toward the target. The idea is that: reward is given when the agent takes the action with direction toward the target position; when the direction of the action is opposite to the target position, a penalty(negative reward) is given from the environment. The azimuth is calculated from the coordinates of the agent and target. Then the reward value is computed as the difference of the azimuths in the current state s_t and former state s_{t-1}. By computing this difference, we can get the direction that the

agent moving towards. The equation is defined as:

$$r_o = arctan\frac{y_{s_t} - y_{target}}{x_{s_t} - x_{target}} - arctan\frac{y_{s_{t-1}} - y_{target}}{x_{s_{t-1}} - x_{target}} \tag{4.4}$$

$(x_{s_t},\ y_{s_t})$ and $(x_{s_{t-1}},\ y_{s_{t-1}})$ are the positions of the robot in the state at t and $t-1$ relatively. $(x_{target},\ y_{target})$ is the target position.

The final reward is the sum of the distance-based reward and orientation-based reward with scaling factors that are adjusted based on the learning efficiency. To adapt to different environments, we designed two orientation-based reward functions. In one case, the reward function given by Equation 4.5 is considered only in case of the robot not being around the obstacle. In the other case, the reward function in Equation 4.6 considers the orientation-based value all the time. The reward functions are defined as the following equations:

$$R(s,a,s') = \begin{cases} 20, & \text{if reach the target} \\ 1 - e^{0.2*d_1} + (-0.5)/d_2, & \text{if } d_2 \leq \text{threshold} \\ 1 - e^{0.2*d_1} + r_o, & \text{otherwise} \\ -20, & \text{if collide the obstacle} \end{cases} \tag{4.5}$$

$$R(s,a,s') = \begin{cases} 20, & \text{if reach the target} \\ 1 - e^{0.2*d_1} + r_o + (-0.5)/d_2, & \text{if } d_2 \leq \text{threshold} \\ 1 - e^{0.2*d_1} + r_o, & \text{otherwise} \\ -20, & \text{if collide the obstacle} \end{cases} \tag{4.6}$$

d_1 is the distance to the target and d_2 is the distance to the closest obstacle. r_o is the orientation difference between the current and the former state.

Potential-based Reward The potential-based reward function defines a potential value $\Phi(s)$ in each position of the environment, based on the distance between the current state s and the target state s' [27]. The potential-based reward value of the current state is defined as $F(s,a,s')$, which is the difference of the potential value of current state $\Phi(s')$ and the value of former state $\Phi(s)$. In addition to the potential-based reward, sparse reward is also taken into account as it in the case of distance-based and orientation-based reward functions. The complete reward value R' equals to the sum of the accumulated potential-based reward value F and the sparse reward value R:

$$R' = R + F \tag{4.7}$$

$$F(s,a,s') = \gamma\Phi(s') - \Phi(s) \tag{4.8}$$

where γ is the discount rate. With this discount rate, the agent will get penalty even though it stays in the same position without increasing the distance to the target. So the agent will be encouraged to explore the path to the target.

The positions besides target and obstacles are set as a value in a range related to the distance to the target. So the reward value will increase as the agent moving towards the target or away from the obstacles. In these cases, an inverse proportional value of the distance is calculated since we expect the potential value to be larger when the agent getting closer to the target, and smaller when the distance to the target getting larger. The value of each position can be adjusted according to the learning efficiency of the experiments. The reward function is defined as:

$$R(s,a,s') = \begin{cases} 20, & \text{if reach the target} \\ \gamma(A/d_1(s') + B/d_2(s')) - (A/d_1(s) + B/d_2(s)), & \text{if } d_2 \leq \text{threshold} \\ \gamma(A/d_1(s') - A/d_1(s), & \text{otherwise} \\ -20, & \text{if collide the obstacle} \end{cases} \tag{4.9}$$

A and B are the scaling factors. A is a positive value as reward is gained when the agent moving towards the target. B is a negative value as penalty is given when the agent moving closer to the obstacles. d_1 and d_2 are the euclidean distances to the target and the closest obstacle relatively.

4.3 RQ2:State Space Compression with Auto-encoder

To sense the complicated and dynamic environment in the navigation problem, an RGB camera is added to achieve detailed information from the environment rather than using only laser sensors. As it is described in Section 2.2.1, the problem brought by a high-dimensional observation such as camera is that the state space input to the neural network will be large and lead to a high computational cost for training. To compress the state space, we apply SRL to map the observation to the lower-dimensional state space. In this case, an auto-encoder mentioned in Section 2.2.2 is applied to learn the main features of the camera observation and the state vector of the compressed state representation is used as the input for DQN. The framework of deep auto-encoder and Q-network we use [14] is shown in Figure 4.2.

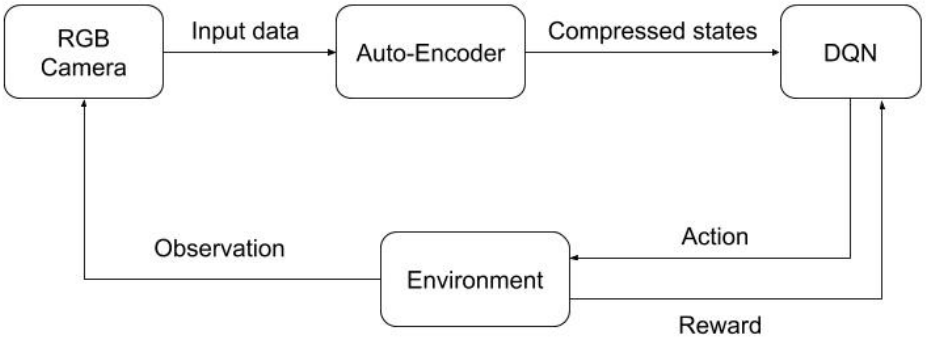


Figure 4.2: Framework for Deep Auto-Encoder and Q-network

The architecture of the auto-encoder with single frame of observation

With the RGB-camera input data, we use a convolutional auto-encoder to compress the observation states and extract the main features of the observation. As shown in Figure 4.3, the architecture of the auto-encoder consists of two convolutional layers and two full-connected layers in the encoder and decoder part relatively. The input layer is a three-dimensional matrix with the size of $32\times32\times3$, where the first two dimensions are the size of the input image and the third dimension represents three channels of the RGB image. The first convolutional layer in the encoder part is designed with the receptive field of 3×3, the stride is 1×1 and the number of the feature maps is set to be 32. Then the output of this layer is $3\times3\times32$. The kernel size of the second convolutional layer is still 3×3 and the number of filters is 64, which means the output of this layer is $3\times3\times64$. Via the flatten layer the three-dimensional output is transferred to a one-dimensional vector in size of 576 and connected to a dense layer with 64 neurons. Then the output is connected to the next dense layer with 10 neurons. The latent state space is the output of the last dense layer in size of 10 and is input to the DQN as a lower-dimensional state vector.

After the encoder part, the decoder part is started with a dense layer in size of 64 connected from the latent state vector. Then the output is connected to the next dense layer and reshaped into $32\times32\times64$, which represents image size$\times$feature maps number. Then it is input to a convolutional layer with 32 filters and kernel size of 3×3, so the size of the output of this layer is $3\times3\times32$. The kernel size of the second convolutional layer is 3×3 and the number of the characteristic plane is 3, which is the number of RGB image channels. After a flatten layer, the output of the decoder is the reconstruction image in the size of $32\times32\times3$.

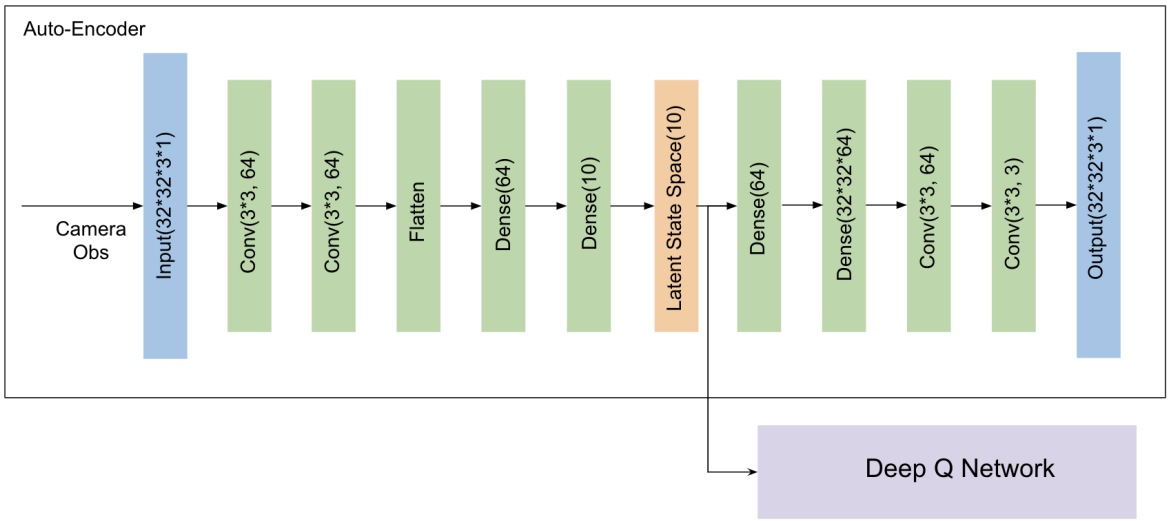


Figure 4.3: Architecture of Deep Auto-Encoder with Single Frame Input

We use the ReLU as the activation function. The loss function for the auto-encoder calculating the loss value between the observation and the decoded image is defined as:

$$Reconstruction\ loss = \sum_{1}^{N}(\hat{obs}_i - obs_i)^2/N, \tag{4.10}$$

where N is the number of data in the training batch; obs_i is the i^{th} camera raw input and $\hat{obs}_i = D(E(obs_i))$ is the reconstruction of the i^{th}camera input.

The architecture of the auto-encoder with sequence frames of observation

In the dynamic environment with randomly moving targets and obstacles, the auto-encoder with single frame of observation struggles to extract features of the motion pattern of these moving objects. To observe the velocity and route of the moving objects and choose a proper action for the next state more precisely, we input a sequence of observations to the auto-encoder in this complex case. Inspired by the image pre-processing method in Atari games [28] [29], we input 4 consequent frames of the observation images to the auto-encoder and encode the features of this sequence of observations in the latent state space. As the limitation of the system memory, we convert the RGB images to grayscale images as the observation. At the beginning of the training episode, we create 4 copies of the first frame of the observation and create a vector by concatenating these 4 images together. So we get a one-dimensional observation vector in size of $32\times32\times4$. Then we add every next observation into the observation vector and input the last 4 frames of observations to the auto-encoder. So we reconstruct 4 frames every time and compute the loss value between the observation and reconstruction states of these 4 frames of

images.

The architecture of the auto-encoder with the sequence of observations is shown in Figure 4.4. Same as the single observation one, the auto-encoder consists of two convolutional layers and two full-connected layers in the encoder and decoder part relatively. The input layer is a three-dimensional matrix with the size of $32 \times 32 \times 4$, where the third dimension represents 4 frames the images. The kernel size and the stride are the same as the case with single observation input. To extract more features as the observation images increases, we double the filter number of the convolutional layers. So the filter number of the first convolutional layer in the encoder part is 64 and the second convolution layer contains 128 filters. The output of the convolutional layers is connected to the first dense layer with the same size as the former case. Then the output is connected to the next dense layer with 15 neurons, which is increased to extract more features for the latent state space.

The decoder part is started with two dense layers connected from the latent state vector with the same structure as the one with single observation. Then it is input to a convolutional layer with 64 filters and the size of the output is $3 \times 3 \times 64$. In the second convolutional layer, the number of feature map is 4, which is the number of observation frames. The output of the decoder is 4 frames of reconstruction images in the size of 32×32.

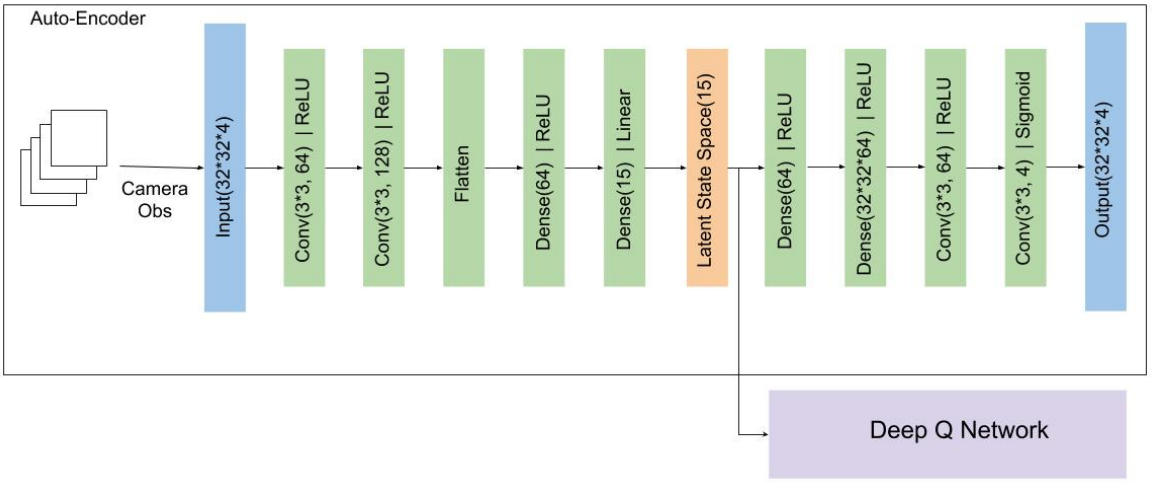


Figure 4.4: Architecture of Deep Auto-Encoder with Multi-frames Input

The training process with the auto-encoder

In our project, the training process with observation compression is presented in Algorithm 2. At the beginning stage of the training process, the actions are chosen by a random policy instead of using an ϵ-greedy policy based on Q-values. Then the Q-network is updated after the first time the state representation updates, which is

after a fixed number of episodes. The number of episodes varies from the complexity of environments in the experiments. In a more complicated environment, we set more episodes between the state representation updates because more samples of observations are needed to extract the main features more precisely. Updating the Q-network and state representation with different frequencies is because the sudden change of the compressed state representation will impact the Q-value distribution and change the policy, which destabilizes the learning [30]. So an alternative updating of the network parameters [31] is applied to reduce this impact.

Algorithm 2 Training process with auto-encoder

Require: Initialize Q network, auto-encoder, $\forall s \in S, a \in A(s)$, first SRL episode e, SRL times n.

if iteration $< e$:
  for each iteration **do**
    Initialize S
    for each environment step **do**
    Observe state s_t and input it to auto-encoder, get the latent space state.
    Input the latent space state as compressed observation, observe reward r_t, and select $a_t \in A$ randomly.
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(a_t, s_t)$
    Update Q network.
    Update states: $S \leftarrow S'$
if iteration == $n * e$ and SRL time $<= n$:
  for each SRL iteration **do**:
    Training every batch of the observations in the auto-encoder, output the encoded states.
    Update state representation
else:
  for each iteration **do**
    Initialize S
    for each environment step **do**
    Observe state s_t and input it to auto-encoder, get the latent space state.
    Input the latent space state as compressed observation, observe reward r_t, and select a_t using policy derived from Q network.
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(a_t, s_t)$
    Update Q network.
    Update states: $S \leftarrow S'$

Chapter 5

Experimental Design

This chapter shows the design of the simulation experiments to verify the performance of the proposed approaches in different environments. We start by setting up the simulation platform in Section 5.1. Afterwards in Section 5.2, we describe different environments used in the experiments and tune the neural network parameters. In the first set of experiments in Section 5.2.1, we aim to validate the effectiveness of the DQN algorithm proposed in [2] with some preliminary experiments. Then in Section 5.2.2 and 5.2.3, we introduce experiments that setting up to answer the research questions in Section 1.2. Then in Section 5.3, we make a brief introduction to test experiments.

5.1 Experimental Setup

The objective of the experiments is to find an optimal route from the original position of the robot to the target with the proposed DQN approach. We use the robot simulator V-REP [32]to build the 3D virtual environment. To interact with the virtual environment, PyRep [33] is used as a toolkit built on top of V-REP. As shown in Figure 5.1, the simulated environment includes cuboid objects as the obstacles and a target for the robot to reach, presented as red circles. In the following experiments, the obstacles are rendered in yellow and blue cuboid objects presenting dynamic and static ones respectively. The pioneer robot is used as an agent and equipped with laser sensors to detect environmental information. The sensors have a 360-degree detection angle range, and 16-dimensional local obstacle distance information can be obtained. The maximum detection distance range of the sensor is 5m. In the experiments in Section 5.2.3, an RGB-camera is added to get an observation image of the environment.

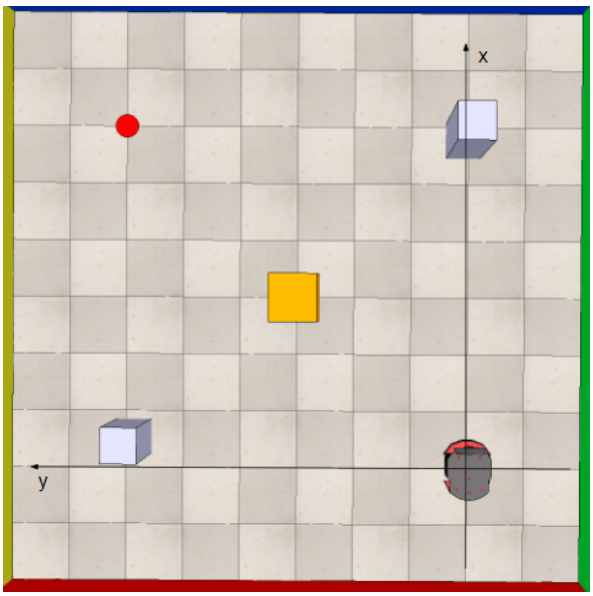


Figure 5.1: Dynamic Environment with one moving obstacle

The DQN network is initialized with a neural network containing three dense layers with 128 hidden neurons that are activated by the ReLU activation function as shown in Figure 4.1. The network receives a state vector containing the information of the environment as the input. The content of the state vector varies in different cases and will be described in the following experiments. The outputs are the predict Q-value for actions in three directions, forward, left, right, with a fixed velocity.

The choice of the action is decided according to the maximum value of all the Q-values for a specific state. To avoid getting stuck at local optima, we apply a decaying ϵ-greedy policy to choose an action at a specific possibility. A discount factor of $\gamma = 0.97$ is used and the batch size is 128. The learning rate τ varies according to the complexity of the environment. The more complex the environment, the less value the learning rate. The training starts with ϵ as 1 to choose an action randomly and then the ϵ value decays from 0.8 to 0.05 in a fixed number of episodes(depends on the complexity of the environment). Since more training episodes are needed in a complex environment, more episodes are needed for ϵ decaying. This ϵ-decay policy aims to choose the action more depending on the learning policy and reflect the improvement of the policy for each time.

Each episode will come to termination when the robot encounters obstacles, reaches the target position or achieves a predefined number of moving steps. We set 200 steps as the max step number of each episode, aiming to train the robot to find an optimal route with limited steps.

All the parameters are shown in Table 5.1:

Parameters	Value	Attribute
Exploration factor(ϵ)	$0.8 \rightarrow 0.05$	Decreases with episodes
Discount factor(γ)	0.97	Constant
Learning rate(τ)	$0.003 \rightarrow 0.0003$	Decreases with the complexity of the environment increasing
Batch size	128	Constant
Maximum steps per episode	200	Constant

Table 5.1: Experiments parameters

5.2 Training Experiments

In every experiment, we have two phases of the training phase and the testing phase. In the training phase, we train the robot with the approaches described in this Section. When the accumulated reward gained by the robot reaches a stable level of positive value, an optimal policy is learned and the training process is successful. After this stage, we will turn to the testing phase to validate the learned policy. The testing phase will be introduced in Section 5.3.

5.2.1 Preliminary Experiments

To validate the effectiveness of the proposed DQN algorithm in implementing the navigation task, a set of preliminary experiments are done with different dynamic environments. We introduce three kinds of environment: moving target and static obstacles, moving obstacles and static targets, moving target and moving obstacles.

In the experiments with dynamic targets and static obstacles environment, as shown in Figure 5.2, three targets are set. The original position of the target is chosen randomly at the beginning of every episode. Taking the original position as the center, the target moves back and forward along the red arrows in a distance range of 3 meters with a velocity of 0.05m/dt. The training episode is 2000 and the learning rate is set as 0.003. The ϵ value decayed to 0.05 from 0.8 in 1000 episodes. The state vector that input to DQN we use in this experiment contains the laser measurements(16) and the position states(coordinates of x and y) of the robot and the target, so the size of state vector is 20.

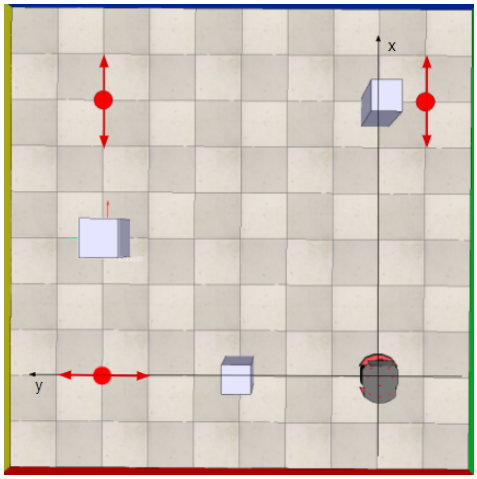


Figure 5.2: Dynamic Environment with moving targets

In the experiments with static targets and dynamic obstacles environment, the moving route of the obstacles are shown in Figure 5.3. The obstacles moving along the clockwise route as presented in blue arrows. The speed of these obstacles are random in range of 0.05m/dt $\sim$ 0.50m/dt. The training episode is 4000 and the learning rate is set as 0.001. The ϵ value decayed to 0.05 from 0.8 in 2000 episodes. The state vector that input to DQN we use in this experiment contains the laser measurements(16) and the position states(coordinates of x and y) of the robot and the obstacles, so the size of state vector is 24.

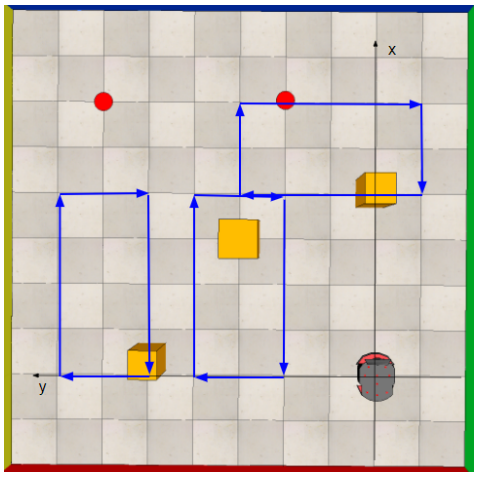
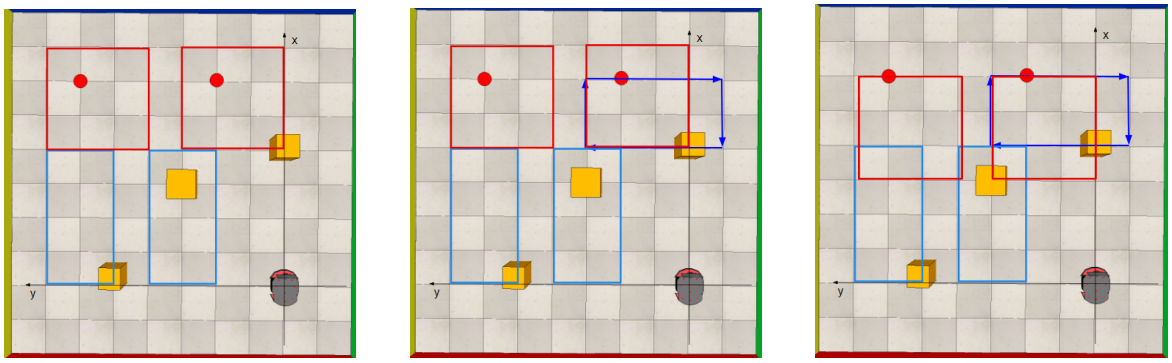


Figure 5.3: Dynamic Environment with moving obstacles

In the experiments with dynamic targets and obstacles environment, we test with three environments in Figure 5.4. Figure 5.4a shows that targets are moving inside the red area with random motion and obstacles moving inside the blue area randomly. The red area and blue area do not overlap with each other. At the beginning of every episode, one of the original positions of the target, rendered in red circles,

will be chosen and the target will move inside the relative area. In the environment shown in Figure 5.4b, it is almost the same as 5.4a but only includes a third moving obstacle with a regular moving motion following the blue arrows. The moving area of this regular moving obstacle has an overlapping area with one of the target motion area. In the third environment(Figure 5.4c), we try with the scene that the motion areas of targets and randomly moving obstacles partially overlap each other. In these environments, the speed of the robot varies randomly from 0.05m/dt to 0.50m/dt in every step. The training episode is 4000 and the learning rate is set as 0.0005. The ϵ value decayed to 0.05 from 0.8 in 2000 episodes. The state vector that input to DQN we use in these experiments contains the laser measurements and the position states of the robot and the obstacles, the size of state vector is 24.



(a) Without overlapping area

(b) Overlapping regular moving area

(c) Overlapping random moving area

Figure 5.4: Dynamic Environment with moving obstacles and targets

The results of these preliminary experiments are presented and discussed in Section 6.1.

5.2.2 Reward Functions

To answer the first research question, we come up with three kinds of reward functions: distance-based, orientation-based, and potential-based as introduced in Section 4.2. In this section, experiments are set to do a comparison among these different reward functions and make a conclusion on the most optimal one for this navigation task. The reward function is designed to avoid the obstacles and approach the target point with the shortest movement path.

The first experiment uses the distance-based reward function containing the exponential Euclidean distance from the robot position to the target position. To find an optimal discount factor γ in Equation 4.1, experiments in the static environment with different γ value are done. Apart from that, the distance to the closest obstacle is considered as an inverse proportional value for obstacle avoidance(Equation

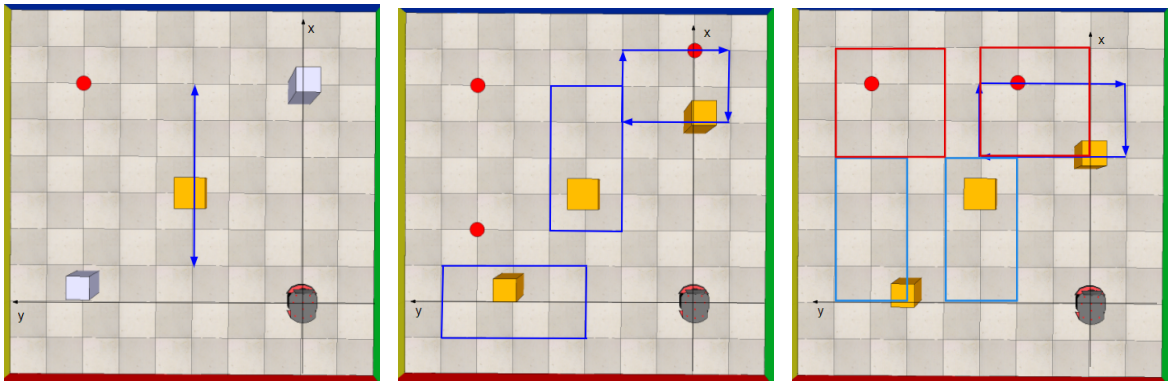
4.3). The second one adds the orientation-based reward(Equation 4.5 and 4.6) that depends on the difference of the azimuths of the robot to the target in two consecutive states. In the third experiment, we apply the potential-based reward that takes the inverse proportional value of distances to the target and obstacles into account(Equation 4.9). We test with different γ value in Equation 4.9 and find an optimal one with experiments in the static target and dynamic obstacles environment(Figure 5.3). We use the parameters shown in Table 5.1 in these experiments. The state vector that input to DQN we use in these experiments contains the laser measurements and the position states of the robot and the obstacles, the size of state vector is 24.

At last, we compare the performance of these reward function in the same environment. In this way, we can answer the first research question with the comparison results discussed in Section 6.2.

5.2.3 Observation Compression

In the experiments to answer the second research question focusing on the observation compression problem, we add an RGB camera to obtain the information from the environment and apply the auto-encoder to extract the main features of the environment. After the auto-encoder generates the latent space representation, the compressed state vector is input to the DQN network and trained to update the Q-values. The reward function we use in this case is the optimal reward function chosen in Section 6.2. The loss function we use in Q-network is Equation A.2. The reconstruction loss in auto-encoder is computed by Equation 4.10.

We set up experiments in three different dynamic environments. The first one contains one single static target with regularly moving obstacles as shown in Figure 5.5a. The motions of obstacles are marked with blue arrows. The second environment includes multiple static targets with randomly moving obstacles as shown in Figure 5.5b. The motion of regularly moving obstacle is marked with blue arrows and moving areas of randomly moving obstacles are marked with blue rectangles. At the beginning of every episode, the target is randomly chosen in a set of predefined target points. We de-visualize all the target points other than the chosen one in the observation to avoid making the robot confused. The third environment has multiple moving targets with randomly moving obstacles in Figure 5.5c. Same as the second environment, the motion of moving obstacles are marked with blue arrows and rectangles. The targets are randomly moving in the area marked with red rectangles. The targets start with two different original positions. One is chosen randomly at the beginning of every episode and the other one is de-visualized. The speed of moving objects is in range of (0.05m/dt, 0.5m/dt).



(a) Experiment 1 *(b)* Experiment 2 *(c)* Experiment 3

Figure 5.5: The environment for visual observation

The camera is set on top of the robot and observes only the part of the environment in front of it. The observation changes when the robot takes action. So the observation reflects the position of the robot at the corresponding state. For easier detection of the target position, we mark it with a red circle(s) at the chosen position(s) in the environment, so the camera can observe it. The colors of the walls in the environment are set to be different to tell apart from different directions.

In the experiments with the first environment(in Figure 5.5a), only RGB-camera is used to observe the environment and the latent space vector from the auto-encoder is input to the Q-network. So the size of the state vector is the size of the latent state space of 10. In the experiments with the second environment(in Figure 5.5b), the laser measurements and some specific states are added because of the increased complexity of the environment. The state vector that input to the Q-network contains the latent space vector, 16 laser measurements, and some additional states including positions of the robot and the three obstacles(we call these states "additional states" in the following context). So the size of the state vector is 34. In the experiments with the third environment(in Figure 5.5c), we apply the sequence of observations in grayscale images for the auto-encoder as introduced in Section 4.3. Only the representation states of the sequence of images are input to DQN. So the size of the state vector is 15. Also, we compare the performances in this environment with the methods using laser measurement, single observation, and sequence of observations.

The resolution of the RGB images from the environment is 640×640 pixels. The images are resized 32×32 before inputting to the auto-encoder. The auto-encoder updates every 400 episodes to collect sufficient samples for the SRL. After testing with different updating times, the state representation will be updated 3 times in the first two experiments to get an encoded state vector containing the main features of the observation. In the third case, as the environment is more complicated, the

update time is increased to 4.

As it is introduced in Section 4.3, an approach is applied with different training frequencies between DQN and SRL to make a balance between the training stability and efficiency. So the auto-encoder updates at the 400th, 800th, and the 1200th(and 1600th in the third environment) episode. When the camera collects information for the SRL before the first update at the 400th episode, the agent chooses the action randomly in this stage instead of applying the ϵ-decay policy. Then the auto-encoder trains with these observations and inputs the compressed latent states of the observations to the Q-network.

To guarantee the accuracy of the encoded states, we set the epoch of the SRL(iteration time) as 20 in single observation cases and 40 in multi-observations cases after tests. In every iteration, the training batch size of the observation in each training epoch is 256. The training epoch in every iteration is 50 with single observation. The number of training epoch is set to be 25 in the multi-observations case because of the limitation of the system memory. The size of latent state space is 10 in single observation cases and 15 in multi-observations cases since the features increased.

After the first update of the auto-encoder, the agent applies ϵ-decay policy to choose the optimal action. The ϵ-decay policy is reducing gradually from 0.8 to 0.05 in 1200 episodes in the first two environments and 1600 episodes in the third environment. This means the ϵ value decays to an invariant value of 0.05 after training 1600 episodes in the first two environments and 2000 episodes in the third environment.

The hyperparameters we use in this section are presented in Table 5.2 and 5.3. Especially, we present the size of the state vector to DQN for every experiment in Table 5.4.

To analyze the performance of the auto-encoder, the decoder will output reconstruction figures after training every 100 episodes. Then we can compare the reconstruction figure with the observation figure in the same state.

The results of these experiments are discussed in Section 6.3 and then we will answer the second research question for state compression in Section 1.2.

Parameters	**Value**	**Attribute**
Exploration factor(ϵ)	$0.8 \rightarrow 0.05$	Decreases in 1600 episodes
Discount factor(γ)	0.97	Constant
Learning rate(τ)	0.0003	Constant
Batch size in DQN	128	Constant
Maximum steps per episode	200	Constant

Table 5.2: Experiments parameters for DQN

Parameters	Value	Attribute
Batch size in SRL	256	Constant
Image size	$32\times32\times3$	With single observation
	32×32	With sequence of observations
Latent state size	10	With single observation
	15	With sequence of observations
Iteration times in SRL	20	With single observation
	40	With sequence of observations
Batch number in each iteration	50	With single observation
	25	With sequence of observations
Episodes between SRL	400	Constant
SRL times	3	With single observation
	4	With sequence of observations

Table 5.3: Experiments parameters for SRL

	Observation	Size of state vector	Content
Experiment 1	RGB camera	10	encoded states of camera image
Experiment 2	RGB camera + laser sensors	34	encoded states of camera image + laser measurements + position states
Experiment 3	RGB camera	15	encoded states of camera images

Table 5.4: State vector for DQN

5.3 Test experiments

In the testing phase after trained successfully with the reward value reaching a stable level, we test the learned policy with 100 episodes in the same environment. The ϵ value is set to be 0, which means the robot will choose the next action only depending on the learned Q value. The average steps that the robot takes in the trajectory to the target and the success rate of the 100 episodes test are taken as evaluation metrics.

Chapter 6

Results

In this chapter, the results of the experiments are presented and discussed to validate the performance of the proposed method and answer the research questions. The curve of average accumulated reward reflecting the learning efficiency and the average loss function value reflecting learning accuracy are shown as results. After training the network, a test is presented for the learned policy will run and the trajectory of the agent.

To analyze the performance of the proposed approaches, the following metrics are set as the main evaluation criteria:

(i) The final Q-value loss calculated from the loss function A.2 after the curve of loss convergences successfully. This evaluates the accuracy that the network able to predict the next state.

(ii) The accumulated reward value after the agent learned an optimal policy, which means the stage after the average reward reaches a stable value.

(iii) The convergence speed presented by the training episodes number that required to convergence to a maximum level of accumulated reward.

(iv) The success rate in the testing phase which is computed when the robot reaches the target in 100 testing episodes.

6.1 Results for Preliminary Experiments

In the preliminary experiments, our objective is to evaluate the performance of the proposed DQN approach in dynamic environments described in Section 5.2.1. The reward function we use in this set of experiments is the distance-based reward function formulated in Section 4.2. In the partial dynamic environments, we use the reward function in Equation 4.2 that is only related to the Euclidean distance to the targets. As the complexity increased, we use the reward function formulated in Equation 4.3 that adds the distance to the closest obstacle in the environments with

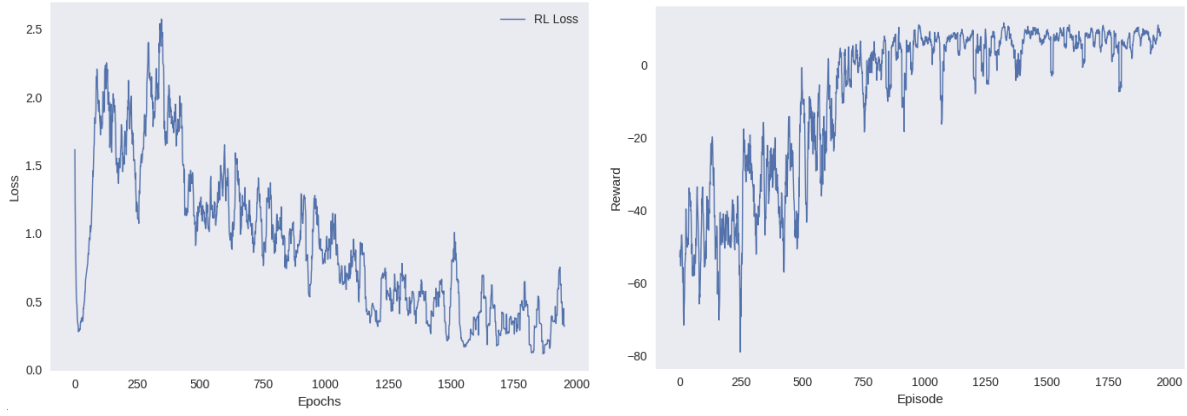
both targets and obstacles moving(in figure 5.4) to help the robot learning the policy.

Results With Moving Targets

In the experiment with moving targets in Figure 5.2, three different targets((-0.5, 3),(3, 3),(3,-0.5)) are set, and one is chosen randomly in every episode. As the moving motion of the targets described in Section 5.2.1, the ranges of the target position are: (-0.5, 3±0.5), (3, 3±0.5) or (3±0.5, -0.5). We use reward function 4.2 in this experiment since the obstacles are static.

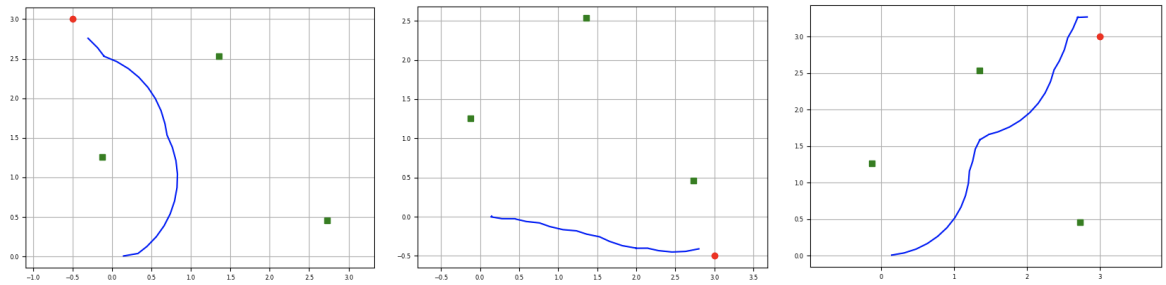
Results in Figure 6.3 show that it takes about 800 episodes for the reward value to reach a level and the robot can reach the target in over 95% episodes after that. The Q-value loss increases to a peak value of 2.5 at the beginning of the training and then convergences to around 0.5 finally.

Figure 6.2 presents the trajectories of the agent achieving different targets in the testing phase. The target is marked with a red circle and obstacles are marked with green squares. The blue lines are the route of the robot toward different targets. In all the situations, the agent can find the shortest route to avoid moving obstacles and reach the target position with a success rate of 100% in 100 testing episodes.



(a) Q-value loss *(b)* Accumulated reward

Figure 6.1: Results in dynamic environment with moving target



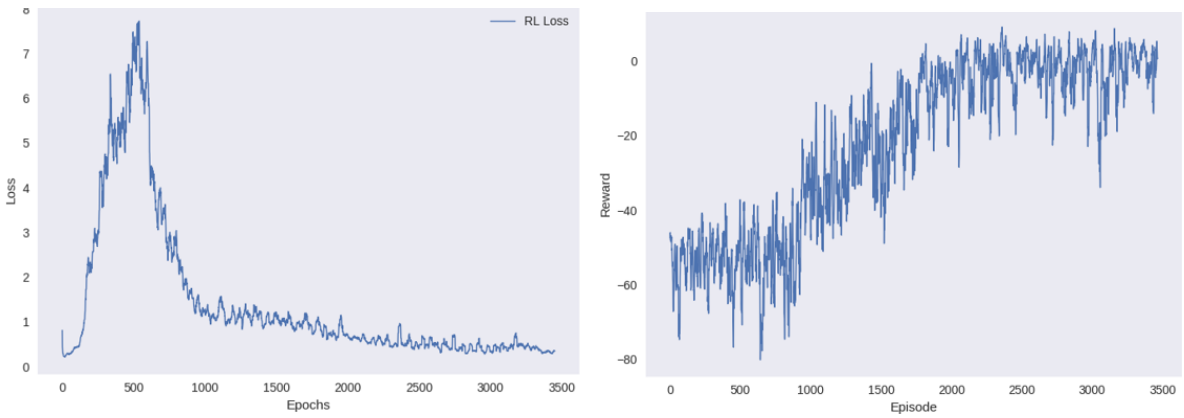
(a) (-0.5,3) *(b)* (3,-0.5) *(c)* (3,3)

Figure 6.2: Test trajectory in the dynamic environment with multiple targets

Results With Moving Obstacles

In the experiment with moving obstacles(Figure 5.3), two different targets((3, 3),(3,1)) are set and one is chosen randomly in every episode. The speed of the moving obstacle randomly changes in the range of 0.05m/dt$\sim$0.50m/dt. In this experiment, we use the reward function 4.3 that considers the moving obstacles.

Results in Figure 6.3 shows that it takes about 2000 episodes for the reward value being stable, where is the exploration factor(ϵ) decaying to 0.05. After 2000 episodes, the robot will reach the target with a success rate of over 90%. The Q-value loss increases at the beginning of training and reaches a peak value during 500 episodes. After that, the loss value decreases and convergences to less than 1 after 1000 episodes.



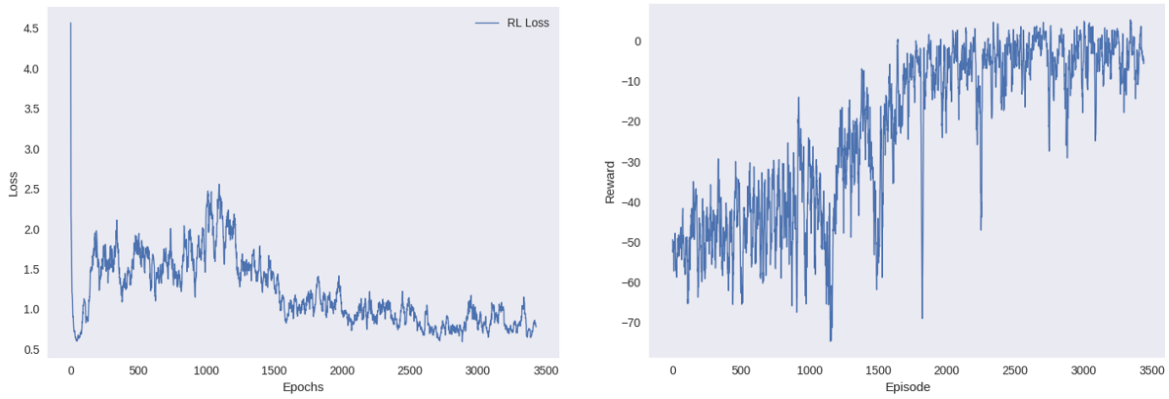
(a) Q-value loss *(b)* Accumulated reward

Figure 6.3: Results in dynamic environment with moving obstacles

Results With Moving Targets and Obstacles

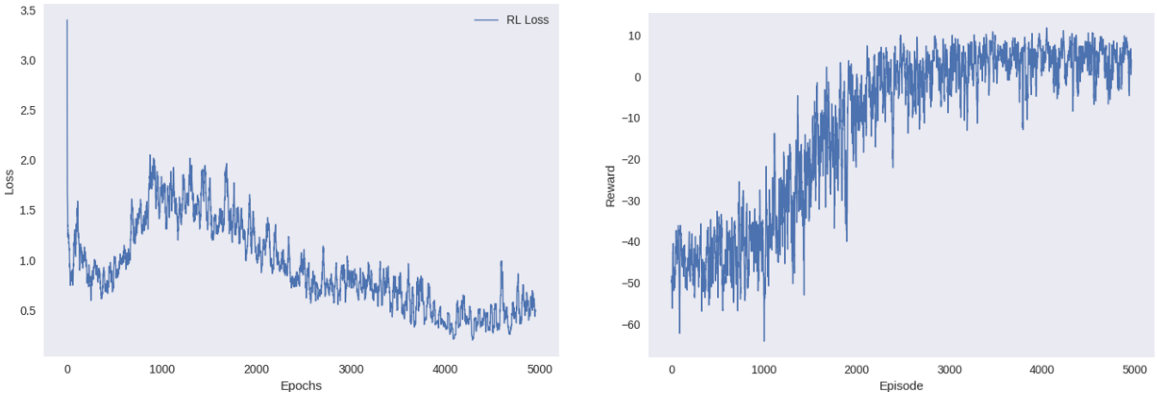
In this experiment, we test in three scenes with different moving areas of the targets and obstacles. We also use the reward function in Equation 4.3 in this case. In

the environment without overlapping moving area of the targets and obstacles(in Figure5.4a) together with the environment with the target moving area overlapping the route of the regularly moving obstacle(in Figure5.4b), the robot will learn the policy successfully after around 2000 episodes as the reward value shown in Figure 6.4b and 6.5b. Figure 6.4a and 6.5a present that the Q-value loss will convergence as the episode increasing.



(a) Q-value loss *(b)* Accumulated reward

Figure 6.4: Results in dynamic environment without overlapping area

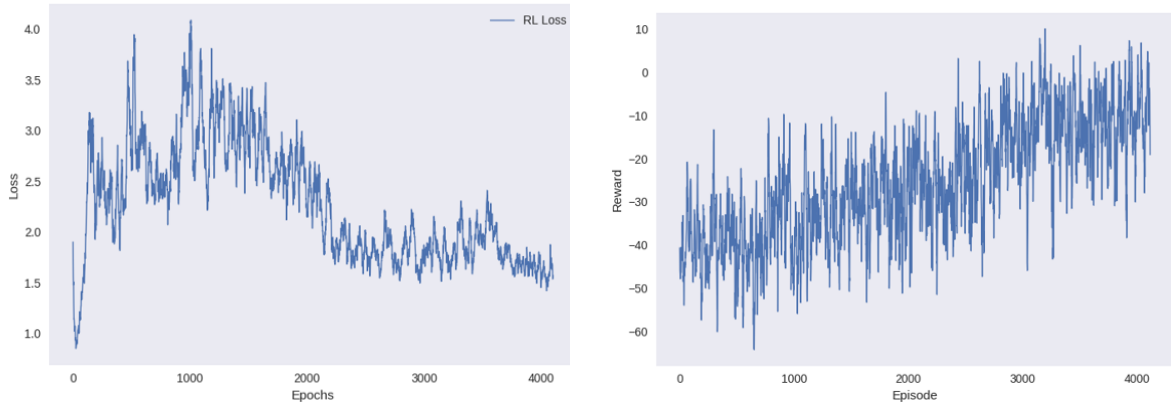


(a) Q-value loss *(b)* Accumulated reward

Figure 6.5: Results in dynamic environment with partial overlapping area

In the environment that the motion areas of the two randomly moving obstacles partially overlap the target moving areas(in Figure 5.4c), the robot fails to learn the policy and reach the target with a high success rate. As the results are shown in Figure 6.6a, the loss value takes around 1000 episodes before decreasing and finally convergences to a high level of over 1.5. As for the reward curve in Figure 6.6b, the oscillation of the reward value is larger than former experiments and it increases with a low growth rate. After the training process is finished, the reward value doesn't reach a relatively stable level and the maximum value is lower than 0.

This should due to the unpredictability of the targets' and obstacles' moving routes. In this environment, there is some overlapping area that the target and obstacle can both enter that area randomly. So the robot is not able to predict that if the target is in the overlapping area, whether the obstacle would run into the area when the robot reaching the target and collide with the obstacle. In this situation, the robot will bypass the motion area of the obstacle and fails to explore a collision-free route. Thus, the robot can only succeed to reach the target when the target is not in the overlapping area and result in a low success rate.



(a) Q-value loss *(b)* Accumulated reward

Figure 6.6: Results in a dynamic environment with overlapping area

In these results of preliminary experiments, a common feature is that the Q-value loss increases at the beginning of training episodes and decrease after reaching a peak value. The reason for this is that at the starting stage, new batches of unknown observations are obtained continuously and the Q-network is not trained enough to predict all the possible states. Thus, the robot is still exploring the environment to find an optimal route to the target and avoid the obstacles. As more information from the environment is obtained, the network will predict the surrounding situation more precisely and the robot will be more confident to explore the route. Another feature is that the reward value has a large oscillation even though the robot has learned the policy. This is due to the random positions of the targets and obstacles in the dynamic environment, so the route and accumulated reward value vary from every episode. Also at the final stage of the training process, some small increment appears in the curve of the loss value when it converging(such as Figure 6.4a and **??**). The reason is that the network can't predict the motion of the randomly moving targets and obstacles, so the oscillation in a reasonable range exists.

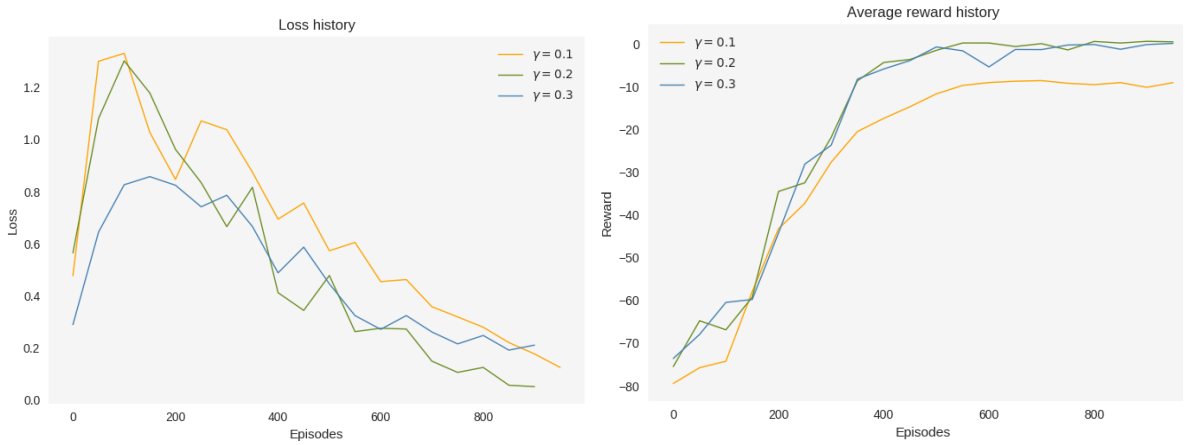
In the following experiments, we will find approaches to improve the learning efficiency and accuracy by testing different reward functions and using the camera to obtain the observation.

6.2 Results for Different Reward Functions

In this section, we evaluate the performance of different reward functions and make comparisons between these reward functions based on the same environment.

Distance-based reward

In the preliminary experiments, we use the distance-based reward function introduced in Section 4.2. For the equation of exponential Euclidean distance(Equation 4.1), the parameter γ is tuned according to the longest distance from the target position to the starting position, which avoids high penalty that results in passive impacts on learning the policy. We test with different values of the parameter γ in the static environment and get the performance in Figure 6.7. We can observe that the accumulated reward value(Figure 6.7b) is higher when γ = 0.2 or 0.3 than the case we use γ = 0.1. But the case with γ = 0.2 convergences to a lower Q-value loss than the cases with either γ is 0.1 or 0.3 in Figure 6.7a. So when the γ value is 0.2, we can get a balanced relation between the penalty in every state and the distance to the target. This means when the robot doesn't reach the goal, the penalty value in every state will not too high to make the agent passive to find the target, or not too low to encourage the agent staying at the same state. So in the all experiments using the reward functions including Equation 4.1 we define the parameter γ equaling to 0.2, such as Equation 4.3, Equation 4.5 and Equation 4.6.



(a) Q-value loss

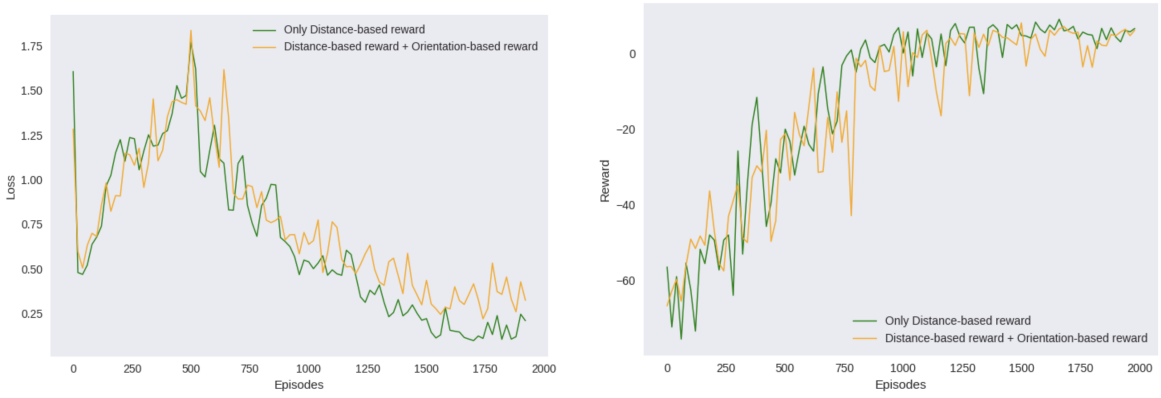
(b) Accumulated reward

Figure 6.7: Performances using different parameter in the distance-based reward function

Orientation-based reward

To explore the impact of the direction taken by the robot in every step, we add the orientation-based reward based on the distance-based reward function. In a simpler environment with a static target and moving obstacle(Figure 5.3), we use the reward function 4.5 with orientation-based reward only in case of the robot not being around the obstacle. This is considering that the obstacle will take avoidance as a prior choice when it moving close to obstacles. To compare with, we also test with the reward function 4.6 that the orientation-based value is considered all the time.

We compare these two reward functions with distance-based reward functions in two different environments. In the static target and moving obstacle environment(Figure 5.3), we only test the Function 4.5 with partial applied orientation-based reward. As shown in Figure 6.8, these two reward functions can hardly tell a difference in the performance. This is due to that the correction of the orientation does not help a lot in this complexity of the environment.



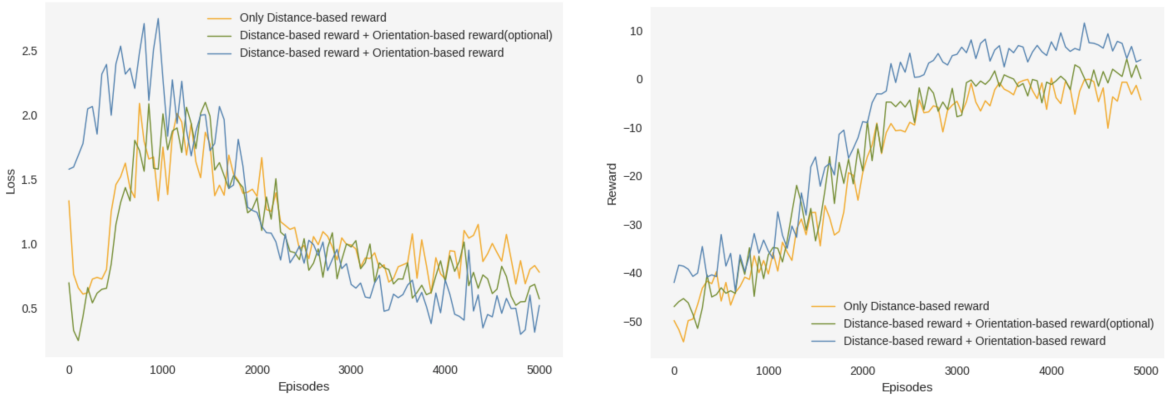
(a) Q-value loss *(b)* Accumulated reward

Figure 6.8: Comparison between reward functions in the dynamic environment in Figure 5.3

Then we tested in a more complicated environment with randomly moving targets and obstacles in Figure 5.4b. We applied the Function 4.6 that considering orientation-based reward all the time and compare the performance of these three reward functions.

As shown in Figure 6.9, the difference between the three reward functions is larger. Function 4.6 performs better with lower loss value and higher accumulated reward. The learning efficiency of these three functions is similar to learning an optimal policy at around 2500 episodes. After testing the learned policy with the exploration factor $\epsilon = 0$, the success rates of these reward functions are also similar at around 94%. So the reward function 4.6 that combined distance-based reward and orientation-based reward performs best in learning accuracy and exploring with

an optimal route to the target.



(a) Q-value loss

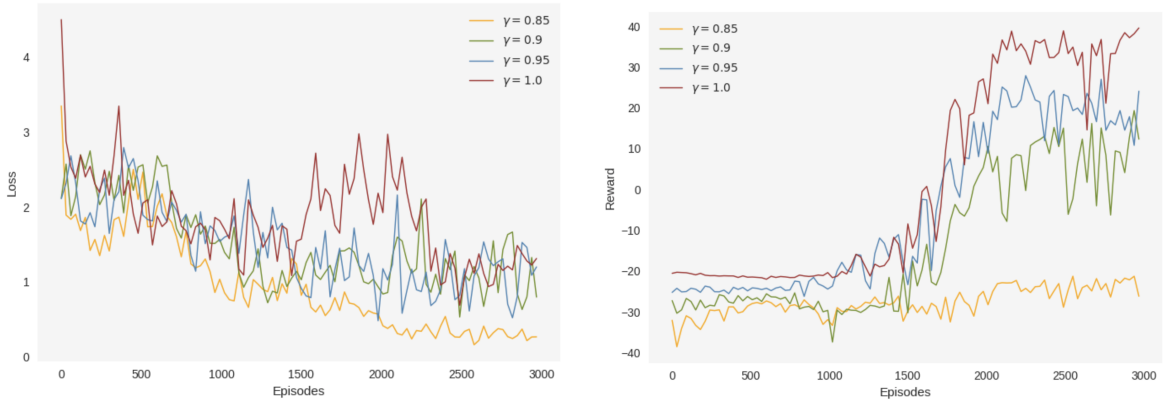
(b) Accumulated reward

Figure 6.9: Comparison between reward functions in the dynamic environment in Figure 5.4b

Potential-based reward

In the experiments using the potential-based reward function presented in Equation 4.9, the parameter of discount factor γ is tuned to a suitable value for the best performance. This parameter determines how much does the potential value of the next state count and how much penalty or reward the agent will get from the potential difference. We choose the environment in Figure 5.3 for testing and four values decreasing from 1.0 are tested.

The curves of loss value and accumulated reward are shown in Figure 6.10 and the success rates in the testing phase are shown in Table 6.1. The function with γ = 0.85 gets the lowest loss but also the lowest reward value. Although the success rate is not low, this is not a proper discount factor value. This result is because the reward value in every step depends more on the former state, it gains less reward towards the target. So the motivation for the robot to find the target is not enough and it takes more steps for the robot to the target with average steps of 150 in the testing phase. When γ = 1.0, the robot obtains the highest reward over the other values, but it only gets a success rate of 65%. The problem is that the potential difference is high. Even though the robot stays in one position, it still gains reward. So the robot tends to rotate in the original position without exploring the environment. The other two γ values have similar performance with a high success rate and reasonable reward value. So the values in the range of 0.9 $\sim$ 0.95 are optimal for the potential-based reward function.



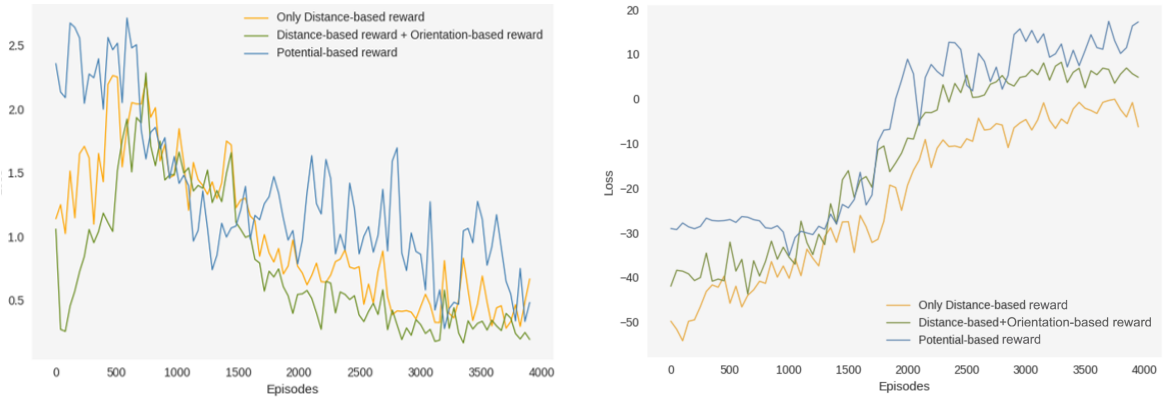
(a) Q-value loss *(b)* Accumulated reward

Figure 6.10: Parameter Tuning

Gamma value	Success rate
$\gamma = 0.85$	88%
$\gamma = 0.9$	96%
$\gamma = 0.95$	94%
$\gamma = 1.0$	65%

Table 6.1: Success rate for different discount factors

After tuning the key parameter of the potential-based reward function. We make a comparison among these three reward functions in the environment from Figure 5.3. From the results shown in Figure 6.11, we obtain the information that the potential-based reward gets the highest loss value and performs unstable with large oscillation in the curves. The reward value with reward function 4.6 is much higher than using reward function 4.3, and also the loss value is lower. The success rate doesn't tell much difference among these three reward functions according to Table 6.2.



(a) Q-value loss *(b)* Accumulated reward

Figure 6.11: Comparison with different reward functions

Reward function	Success rate
Distance-based	93%
Orientation-based	94%
Potential-based	96%

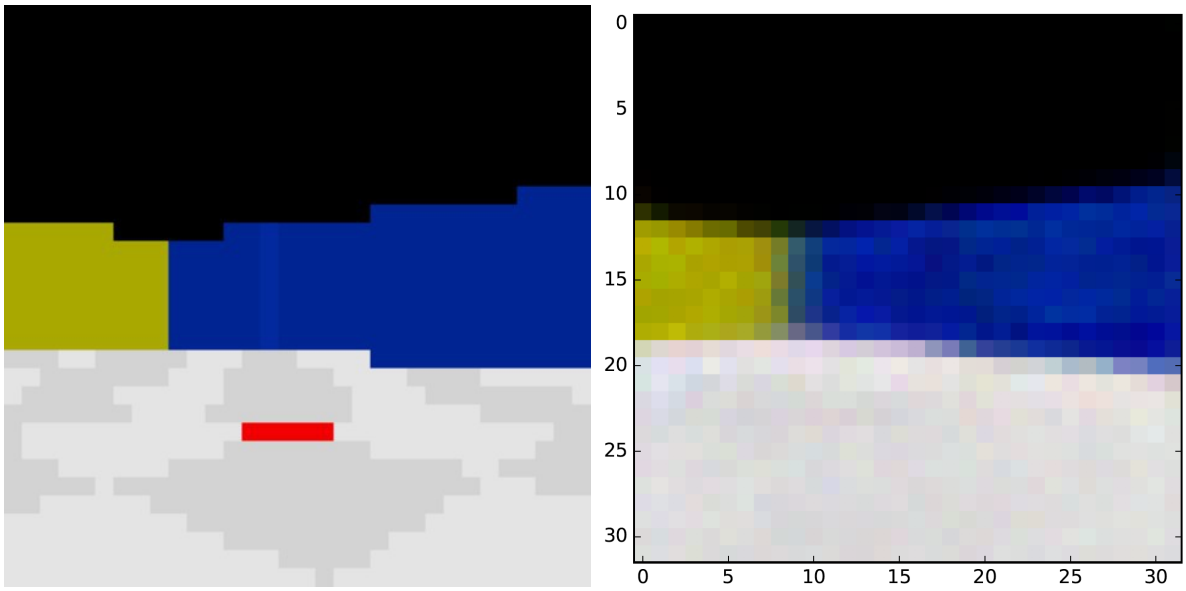
Table 6.2: Success rate for different reward functions

6.3 Results for Observation Compression

In this section, we discuss the performance after adding a camera and applying the auto-encoder to compress the camera observation to states in different dynamic environments. Except for the evaluation criteria described at the beginning of this chapter, the performance of the reconstructed image from the decoder is considered as an auxiliary metric. In the experiments in this section, we use the reward function 4.6 as the optimal one chosen from Section 6.3.

Evaluation of the auto-encoder

In this experiment, we test the performance of the auto-encoder to check whether the important features of the environment are encoded in the latent space state. We test in the static environment with different target positions. Every target position will change randomly at the beginning of the episode.



(a) Observation *(b)* Reconstruction

Figure 6.12: Observation and reconstruction images in the same state

After the auto-encoder is updated 3 times, we can observe in the reconstruction

image in Figure 6.12b that the walls are well reconstructed but the red target point in Figure 6.12a is missed.

To find out whether the target position is encoded in the latent space state, we plot the trajectories to two different targets after the robot has learned the optimal policy. Then we plot the state of every step in these trajectories in the PCA image for the latent space state. To compare the states, we select two similar observations in the routes toward different targets.

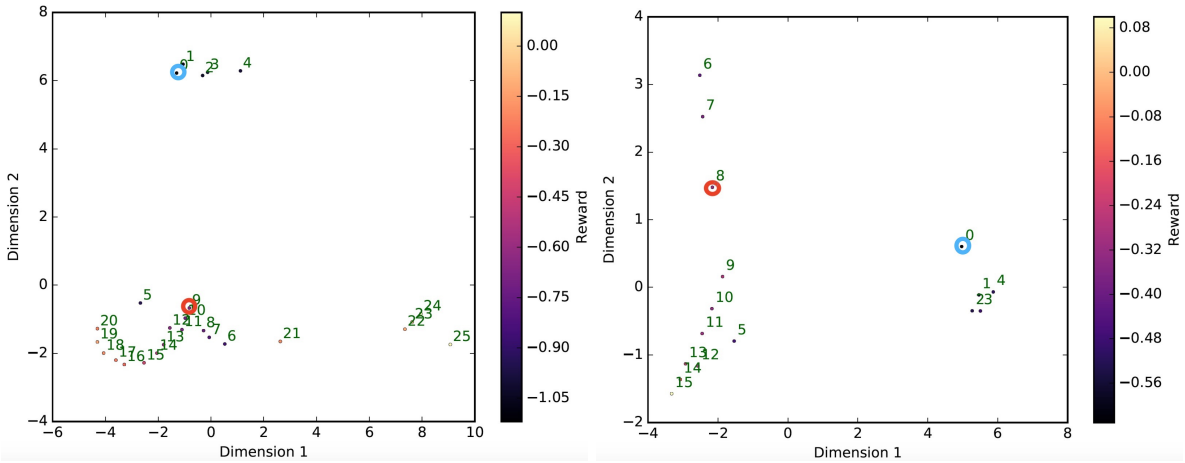
(a) The original state observation in both routes

(b) The observation contains target position in one route

(c) The observation contains target position in the other route

Figure 6.13: Similar observations in different routes

In Figure 6.13a we select the origin position that the robot in different cases observes the same scene. In Figure 6.13b and 6.13b, we choose two similar scenes contain the red target point in the two different routes. The PCA plots in Figure 6.14 shows the clustering of states in the trajectories. The red circles are the states of the targets and the blue circles are the states of similar observations in different routes. We can observe that even in similar positions in the environment, the state representation of these states are different from each other when the targets are different. This means that the state representation of the observations contains the information of the target position by telling the difference of the targets.



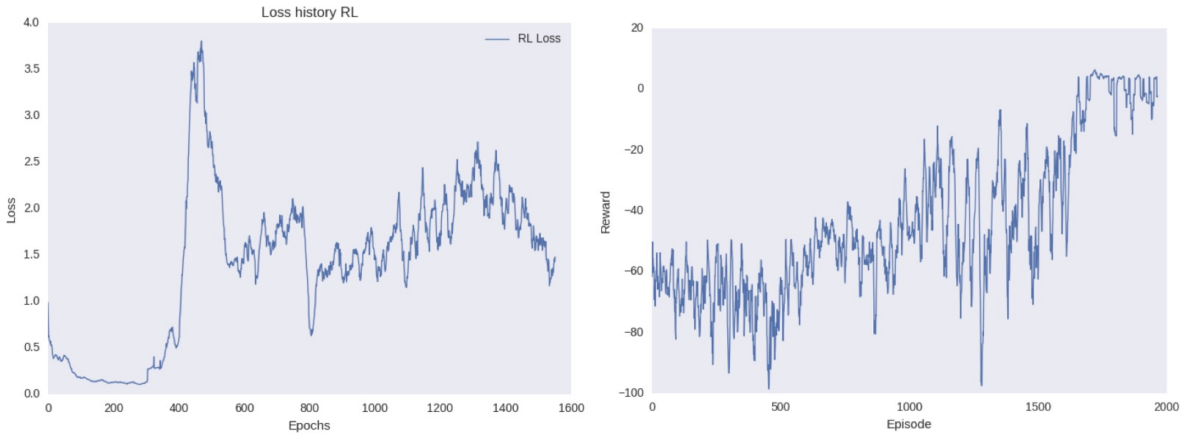
(a) The pca plot represents the state of the observation in Figure6.13b

(b) The pca plot represents the state of the observation in Figure6.13c

Figure 6.14: The pca plot represents the state of the observation to different target

Experiment 1

In the first experiment with a single static target and regularly moving obstacles(Figure 5.5a), the camera is set on the robot observing the front view.



(a) Q-value loss

(b) Accumulated reward

Figure 6.15: Results in the environment with single target and regular moving obstacles

The curves of loss value and accumulated reward are presented in Figure 6.15. The reward curve shown in Figure 6.15b starts increasing after the first time the auto-encoder updates at the 400th episode and reaches a stable level after training around 1700 episodes, which means it learns an optimal policy after the ϵ value decayed to 0.05.

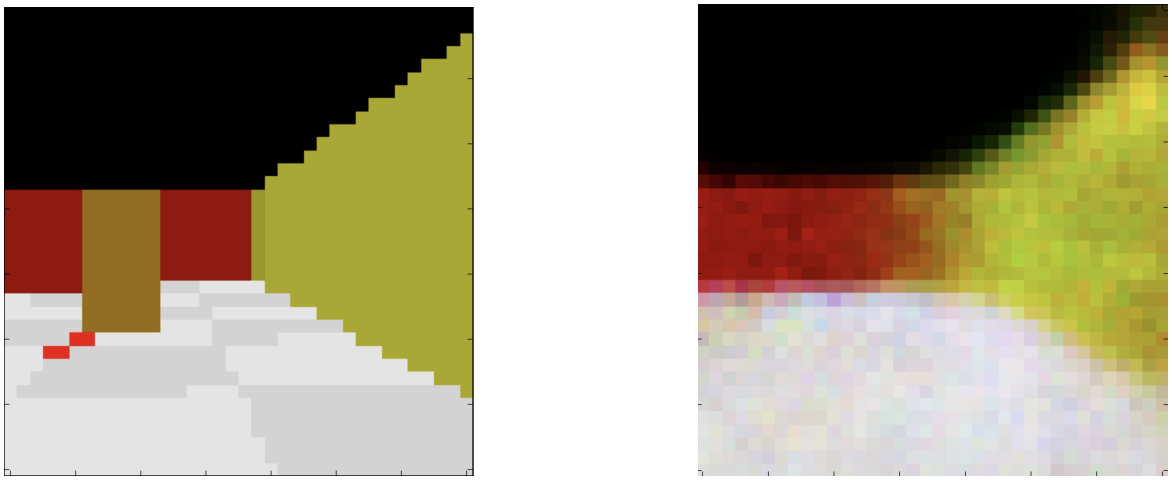
The reward value presents a sudden decrease after every time updating the state representation and after that the reward value increases. The loss value shown in

Figure 6.15a also increases suddenly after the first and second time the state representation updates. This is because of the sudden change of the state representation and the main features of the observation are not extracted accurately at this stage. During the training process between the auto-encoder updates, small increases appear in the loss value curve due to the new samples from the observation. The loss value convergences after the auto-encoder finished 3 times updating, where the encoded states contain the main features of the observation.

The success rate at the testing stage is 93% with an average of 24 steps per episode, which indicates the optimal policy is learned in this case the agent observes the environment through a front-view camera on the robot.

Experiment 2

In the second experiment with multiple static targets and randomly moving obstacles in a fixed area(in Figure 5.5b), we still use the camera on the robot with the front view. The target is chosen randomly in the target set at the beginning of every training episode. However, the robot failed to learn a policy when training with the same state vector and parameters as it in Experiment 1. The reconstruction image also misses the important information of moving obstacles, such as Figure 6.16. This is because when the obstacle is moving randomly in a fixed area and the area is on the way to the target, the robot with a front view camera can't predict the motion of the obstacle and be aware of it if the obstacle runs into the robot from the directions out of sight.

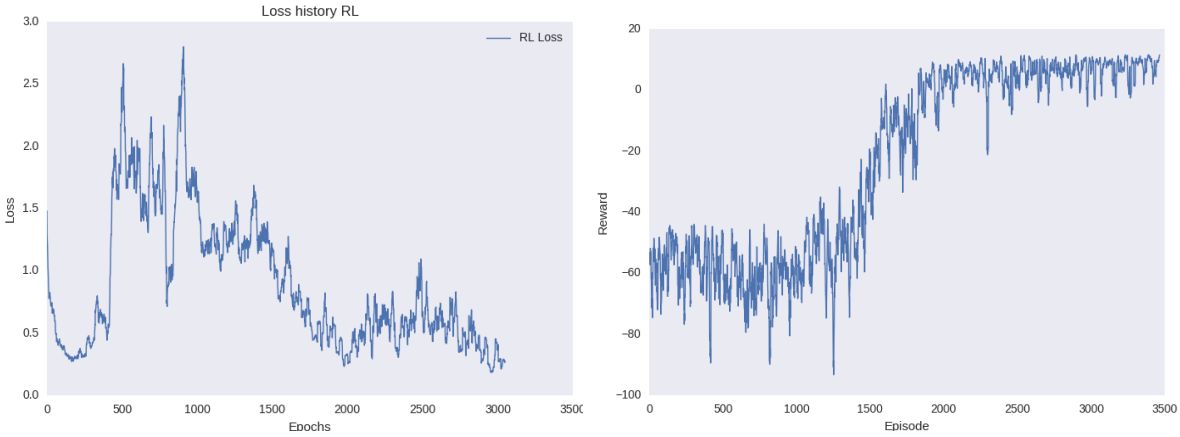


(a) Observation *(b)* Reconstruction

Figure 6.16: Observation and reconstruction images in the same state

To solve this problem, the laser measurements together with a set of additional states are added to the state vector. The additional states include the position of the

robot, the orientation of the robot, and the positions of the obstacles. So the agent is aware of the positions of the obstacles and the distances to them. So the state vector that input to DQN includes the latent state space encoded from the camera observation, the laser measurements, and the position states.

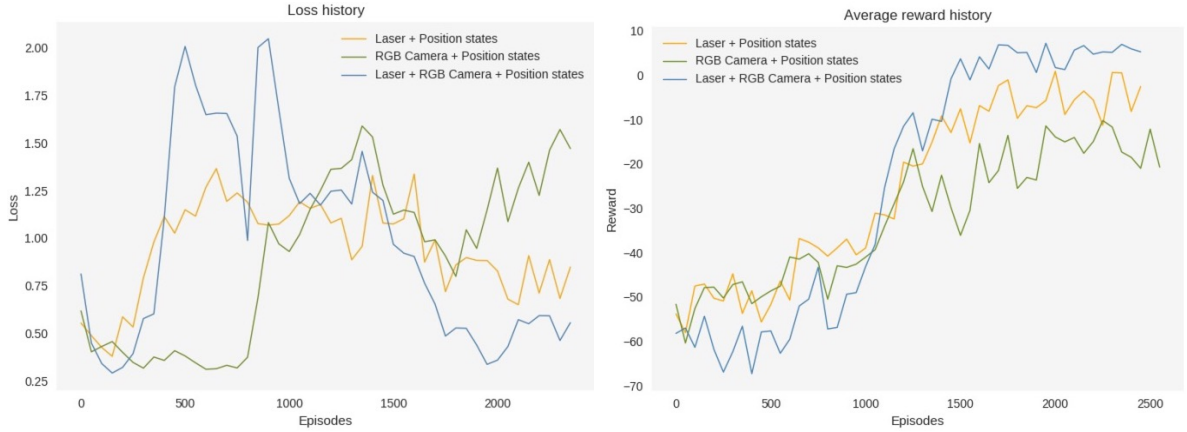


(a) Q-value loss *(b)* Accumulated reward

Figure 6.17: Results in the environment with single target and regular moving obstacles

As the results shown in Figure 6.17b, the reward begins to increase after the first time the auto-encoder updates at the 400th episode and reaches a stable level after training around 1900 episodes. So an optimal policy is learned with an average reward value of 5. In the training process, before the policy is learned, the reward values fluctuate greatly while increasing. This is because of the change of targets in every episode that leads to different routes with variant accumulated reward. The random motion of the obstacles also contributes to these fluctuations because of the uncertainty of their state.

Similar to the results in Experiment 1, the Q-value loss shown in Figure 6.17a increases suddenly after the state representation updates because of the sudden change of the state representation and then decreases because of the training of the Q-network. Also, the impact of this change reflects in the curve of reward with 3 times sudden decrease after the 400th, 800th, and 1200th episodes. The loss value decreases and convergences under 0.5 after the auto-encoder finishes 3 times updating. There is a small increase after episode 2000, which is because of the uncertainty of this dynamic environment with randomly moving objects. The appearance of some new samples of the observation will lead to a small oscillation in the curve.



(a) Q-value loss *(b)* Accumulated reward

Figure 6.18: Comparison between the results of different observation data

A comparison of the results among cases with different observation sensors is presented in Figure 6.18. All these cases include the state of the positions in the state vector. The loss value with only camera observation keeps at the highest level and the laser+camera case has the lowest loss value, although the latter one has a high loss value before the auto-encoder finishes updating(in Figure 6.18a). This shows that the camera observation helps for higher accuracy in choosing a proper action than the cases with only laser sensors. For the convenience to compare with, we cut the part of the reward curve before the first time update of the auto-encoder in the cases with camera observation. As shown in Figure 6.18b, these three methods have a similar efficiency to learn an optimal policy in around 1500 episodes. However, the approach with both laser and camera observations reached a higher reward value than the other two cases. The policy with laser and camera observation can accumulate reward over 5 with 24 steps on average reaching the target. In comparison, the policy with only laser measurement gets a reward of around -5 with over 30 steps on average to achieve the goal. The policy with only camera observation performs worse with negative accumulated reward value. And also the reward values are not stable in this stage.

State vector	Success rate
Laser measurements + position states	92%
Camera observation + position states	64%
Laser + Camera + position states	96%

Table 6.3: Success rate with different state vectors

When we turn to the testing phase, we can get a success rate of 92% in the case with only laser sensors and 96% with both laser sensors and camera from Table 6.3. As the higher dimension of data is received from camera observation than laser

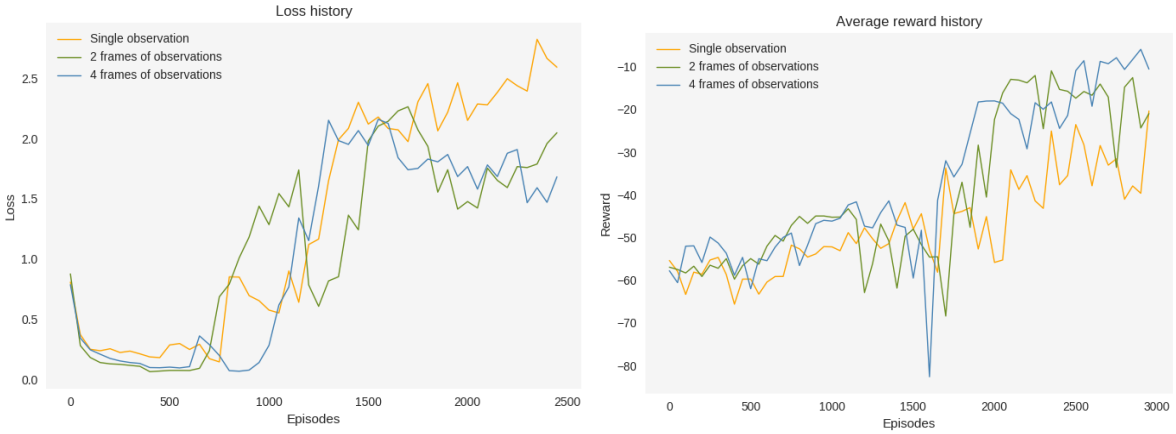
sensors, the agent can learn more information with sufficient samples and building a more complete knowledge from the environment. So the visual sensors lead to a more efficient route to the target. However the case with only the camera has a success rate of only 64%, so including the laser measurement and additional position states in this dynamic environment helps to get accurate positions of the obstacles from the environment and improve the performance significantly.

Experiment 3

In this experiment with randomly moving obstacles and targets in a fixed area(in Figure 5.5c), the motion pattern of the moving objects is hard to predict because of the high uncertainty. We make experiments with two methods to extract more important information from the environment and improve training performance.

The first one is the method we use in Experiment 2 that adds position states in the state vector to get the accurate position of the obstacles and the robot in every step. So the distance from the robot to the moving obstacle is known by the agent and can be used for obstacle avoidance. But as more states are input to DQN for training, the simulation time, in this case, is 2 times as the case without additional states. The second method is to input 4 frames of consequent observations as it introduced in Section 4.3 and 5.2.3. With the consequent observations in a sequence of timesteps, the agent can observe the motion pattern of the moving objects. Then it makes a path planning to avoid moving obstacles and find the moving target by predicting the route of these moving objects.

In Figure 6.19, we present the comparison result among three different number of input observation frame: single, 2 frames, and 4 frames. It's obvious to observe that the case with 4 frames of observations has the best performance. It has the lowest Q-value loss of 1.5 in Figure 6.23a at the final training stage and convergences fastest compared with the other two cases. Figure 6.23b presents that it reaches the highest reward of -10 with the highest efficiency and tiniest oscillation in the final stage. The case with single observation gets the worst performance with the highest Q-value loss of 2.5 and the lowest reward value of -30.



(a) Q-value loss *(b)* Accumulated reward

Figure 6.19: Comparison between the results of different frames of observations

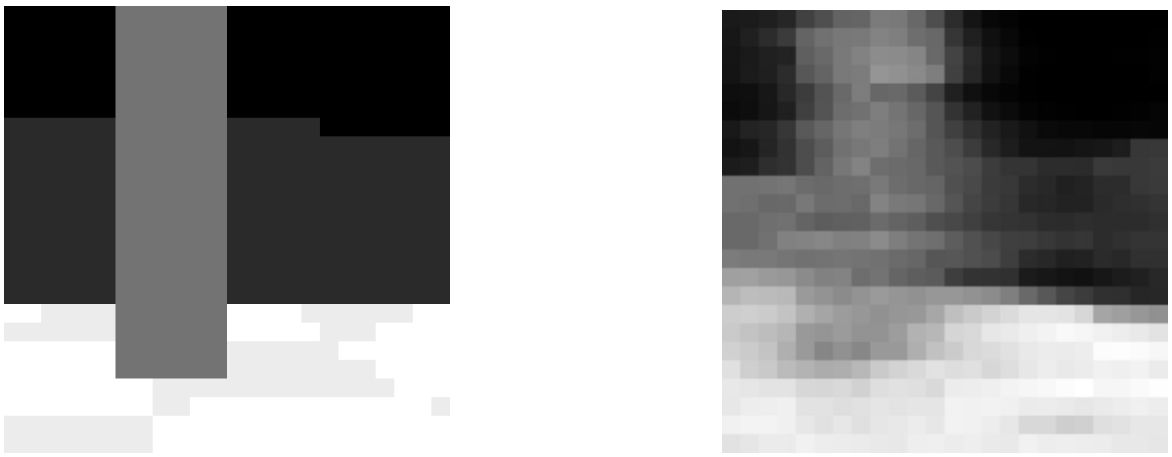
The observation and reconstruction images in single frame and 4 frames of observation cases are presented in Figure 6.20 and 6.21 relatively. We can notice that with single frame observation, the reconstruction image is totally different from the observation image. This is the same reason as the failure of reconstruction in Experiment 2(Figure 6.16). As the environment becomes more complicated, more important information is failed to be extracted with a single observation.

(a) Observation *(b)* Reconstruction

Figure 6.20: Observation and reconstruction images with single observation

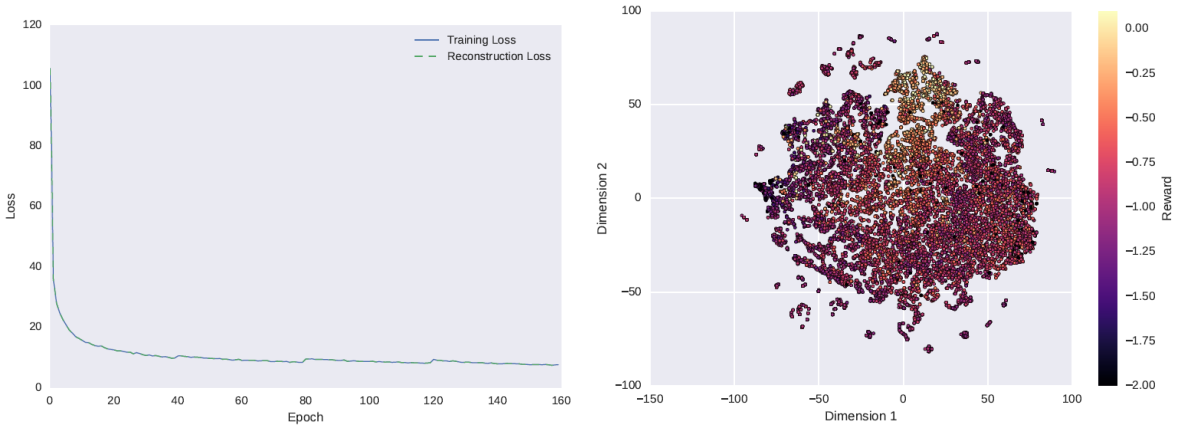
When we come to the reconstruction image from sequence of observations in Figure 6.21, we can find that the reconstruction image is blurry. This is because of the limitation of the system memory, the training samples for auto-encoder are not enough so the information is lack. The other reason is that the obstacles are moving randomly, therefore it is impossible to predict their positions exactly. But the position information of the obstacles can still be extracted by detecting its contour. The re-

sults of latent state space in Figure 6.22a shows that the reconstruction loss value convergences to lower than 5 after four times SRL, which means the high similarity of the main features between the observation and reconstruction images. Figure 6.22b presents the clustering of the states related to the reward value. We can observe from the obvious clustering colors that the SRL can encode and cluster the states with similar rewards value together. So the robot learned enough information for the navigation task from the encoded states.



(a) Observation *(b)* Reconstruction

Figure 6.21: Observation and reconstruction images with 4 frames of observations



(a) AE loss value *(b)* T-SNE visualization

Figure 6.22: Results in latent state space

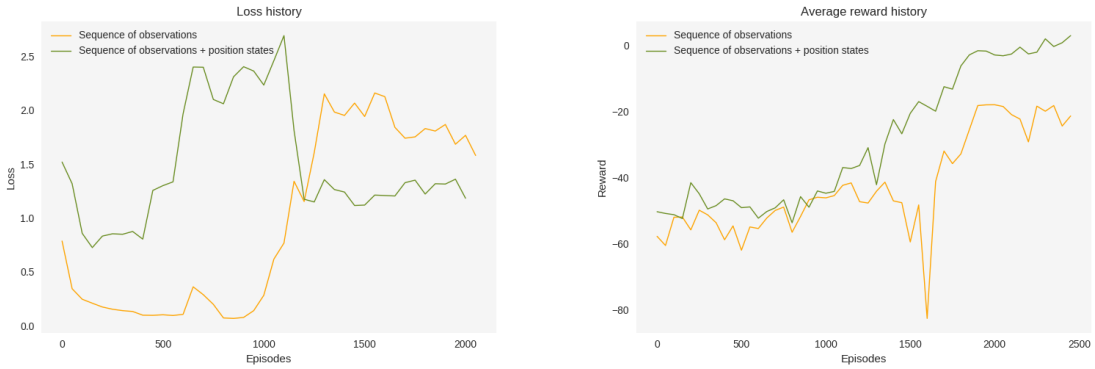
In Table 6.4, we present the success rate with different frame numbers of observations in the testing stage. With single or 2 frames of observation, the success rates are lower than 50% which means the agent didn't learn an optimal policy without enough information from the environment. The case with 4 frames of observations reaches the success rate of 99%, so 4 consequent observations are enough for the

auto-encoder to extract main information from the environment to train an optimal policy.

Frame number of observation	Success rate
Single	33%
2 frames	41%
4 frames	99%

Table 6.4: Success rate with different frame numbers of observation

Then we compare the results with these two methods using in Experiment 2 and Experiment 3: encoding the states of single camera observation and then inputting to DQN with laser measurements and additional states, encoding the states of 4 frames of consecutive observations. In Figure 6.19, we can observe that the first method with laser measurements can convergences to a lower Q-value loss and reach a higher reward which means it can find a better policy. However, since the size of state vector increases with the additional states in the first method, it will increase the computational complexity and take more time to train an optimal policy.



(a) Q-value loss

(b) Accumulated reward

Figure 6.23: Comparison between the results of different observations

Chapter 7

Discussion

In this chapter, we analyze further about the results in Chapter 6 and the performance of the approaches mainly considering the two metrics: the Q-value loss and the accumulated reward.

7.1 Preliminary experiments

In Section 6.1, we conduct the experiments with DQN in different dynamic environments to evaluate the performance of this approach. According to the training results and high success rate in the testing phase, the DQN approach is effective in this navigation problem. As for the experiment in a regular moving obstacles environment(in Figure 5.4a), the robot learns to predict the route of the obstacles and bypass them when it moves close. As for the experiment in a randomly moving obstacles environment(in Figure 5.4b), the robot will avoid entering the area that the obstacles moving in. When the target is moving, the robot will track the regular moving route or enter the randomly moving area to find the target. If the motion areas of the target are accessible and that of obstacles are predictable, the robot will find a collision-free path to reach the target. To learn the optimal policy, auxiliary states include the positions of the robot and the target, the distance between the robot and the target are considered and predicted by the Q-network, which are crucial for the success of the training process. With these additional states, the agent is aware of the position of the robot and estimate the distance from the target to plan the route.

7.2 Reward Functions

In Section 6.2, we present the results of different reward functions based on the distance to the target and obstacles, the orientation of the robot toward the target, and the potential value of the robot. In comparison between reward function 4.3

and 4.6, the latter one also includes the orientation of the robot, the performance of these two reward functions are similar in a simpler environment such as static target and moving obstacles. As the complexity of the environment increases, the advantage of the reward function 4.6 combined both distance and orientation factors will expand (in Figure 6.11). This is because in an environment with more dynamic objects, a more precise prediction of the states need to take more auxiliary measurements into account. So a better policy is learned with higher reward and shorter routes. As for the results in Figure 6.11, we didn't compare the accumulated reward value of the potential-based reward function with the other two, since the calculation and the ranges of the reward value are different according to Equation 4.9. But we can observe from the trend of the curve that the potential-based reward doesn't perform stable compare to the results of the other two functions and also has a higher loss value. Due to the dynamic environment, the potential value of every position is always changing as the obstacles and targets moving randomly, the difference between two consecutive states cannot present the same potential value even though the robot is in the same position. This increases the uncertainty of the reward value in the training process and results in the oscillation.

7.3 Observation Compression

Section 6.3 presents the performances after using an RGB camera with higher dimensional observations and applying the auto-encoder to learn the state representation of the camera images and extract the main features from the observations. The reconstruction images and the visualization of latent state space presented in this section show the performance of the observation compression. Although we can't find the target point in the reconstruction images (in Figure 6.12), the features of the target position are encoded. Since when we plot the state of every step in the trajectories to different targets, the states with similar observations are mapped in different positions of the plots (in Figure 6.14). This means that the change of the target position tells a difference in the state representation of the environment. The target positions are extracted in the latent space state and input to the DQN for training. So in this method the auto-encoder helps to compress the high-dimensional observations to low-dimensional vectors and extracts the main features of the observations successfully.

In the environment with regular moving objects(in Figure 5.5c), the agent can learn an optimal policy successfully with only a single RGB-camera observation and the auto-encoder. But when it comes to more complicated cases with randomly moving objects and targets, the single observation image is not enough for the robot to learn the main information since the position of the obstacles is hard to predict.

We come to two methods: combining the encoded states of the single observation with laser measurements and adding additional information, inputting a sequence of observations for feature encoding.

Adding additional information such as positions of the robot and the obstacles helps obstacle avoidance and makes up for the limited sight. We compare the performance with three different observations: laser sensors, RGB camera, laser sensors together with RGB camera. All these three cases are combined with the additional information mentioned above. From the results in Figure 6.18, we notice that the case combined with both laser sensors and the RGB camera get the highest accumulated reward value and also the lowest loss value. Also in the testing phase, the case of the combined sensors gets the highest success rate. This means that with the help of laser measurements and position information, a higher-dimensional observation helps the robot to find a better policy than using only low-dimensional observation.

The second method, input a sequence of observations to auto-encoder, can help to extract information of moving route and velocity of obstacles and targets. We compare the performance of different numbers of input frames in Figure 6.19 and the case with 4 frames of observations gets the best performance with the highest reward and success rate(Table 6.4). Compared to single observation, sequence of observations records the motion pattern of the obstacles and targets in the field of view. This enhances the success rate for reaching the target in a dynamic environment. According to the t-SNE plot of latent state space in Figure 6.22, the robot can learn the meaningful information from the environment and find a route to the target based on the extracted information after the SRL with 4 frames of observations.

When comparing with these two methods in Figure 6.23, we notice that the first method performs better in converging to lower loss and accumulating higher reward. This is because the laser measurements and position information position the obstacles and robot precisely in every time step. Although the agent can obtain the motion pattern of the obstacles from a sequence of observations in the second method, the location of the obstacles can't be predicted precisely because the moving routes are random and result in a less optimal result than the first method. However, adding the position states in the state vector, of which the size is over two times as the size of the state vector in the second method, will increase the simulation time significantly. In the real world, it is impractical to obtain the exact position of the obstacles in the environment all the time. And using extra laser sensors in addition to RGB camera also increase the cost but not improve the performance obviously according to the success rate in Table 6.3 and 6.4.

Chapter 8

Conclusions and Future work

In this chapter, we first explain how we answer the research questions that we raise in Chapter 1 with the conclusions of our experiments in Section 8.1. Then we discuss the future work for this project in Section 8.2.

8.1 Conclusions

The goal of this thesis is to achieve the navigation for the robot in a dynamic environment. The solution is realized by training the DQN to learn an optimal policy. It is proven that the robot could learn a policy to plan the shortest path to the stochastic dynamic target and avoiding moving obstacles by choosing an action from the maximum Q-value to gain an optimal reward value. After analyzing the results of the experiments, we could answer the formulated research questions:

- RQ1: *What is a proper reward function for the navigation task in a dynamic environment to help the agent learning the policy and accelerate the learning efficiency?*

  To answer this question, three kinds of reward functions are tested in the experiments: distance-based, distance and orientation-based, and potential-based. In Figure 6.8, the reward function based on both distance and orientation performs better with higher reward value and lower loss than the one based only on distance. As for the results in 6.11, the success rate and reward value of these three reward functions are similar in a partial dynamic environment. The first two reward functions perform better than the potential-based one because the loss value of the latter one is higher and the reward curve in the final stage has large oscillation which dues to the uncertainty of the potential value. Therefore, among the tested reward functions, the optimal one that can be applied in a dynamic environment is the distance and orientation-based one in Equation 4.6.

- RQ2: *In the navigation problem, how to reduce the dimension of the state space and extract meaningful information with auto-encoder in a dynamic environment?*

  To reduce the dimension of the observation state and extract main features from the environment, we trained the auto-encoder and Q-network alternatively and the ϵ-decay policy is applied for taking an action after the first time updating the state representation. As the comparison results given in Figure 6.18, the case with state compression can reach a higher accumulate reward and plan a shorter path to the target than the case with low-dimensional observation such as laser measurements. As the results of Experiments 2 and 3 shown in Section 6.3, by adding position states or applying sequence of consequent observations in the auto-encoder, the motion pattern of moving objects can be encoded and the robot can succeed in reaching the target in a complicated dynamic environment. In these cases, high-dimensional data with more information from the environment can be applied to extract the task-relevant knowledge, which is beneficial for the agent to learn an optimal policy with the shortest route. Considering a shorter simulation time and practicality in the real world, using the method with only RGB camera and learning the state representation from a sequence of observations in auto-encoder is preferred in this navigation task.

8.2 Future work

In the experiments for observation compression, we use a sequence of observations to extract more information from the environment. But due to the limitation of the system memory, we convert RGB image to grayscale image and use a limited size of memory buffer as well as latent state space in the auto-encoder. In the future, we will increase the size of the observation image, memory buffer, latent state space, and the number of filters to extract more meaningful information in the dynamic environment. So the performance can be improved.

In our project, we test the learned policy only in the same environments as we training it. In future work, a generalization of the policy will be implemented using transfer learning methods, so the learned optimal policy can be generalized to a larger environment, with obstacles moving in different areas or with more moving obstacles.

The navigation of the robot is expected to be able to be realized in the real world. We only implemented all of the proposed methods in virtual environments. We will apply transfer learning to generalize the policy learning on the simulation platform

to the real robot and environment in the future. This can avoid expensive time and human costs from learning a policy directly from the real environment.

Bibliography

[1] T. Lesort, N. Diaz Rodriguez, J.-F. Goudou, and D. Filliat, "State representation learning for control: An overview," *Neural Networks*, vol. 108, 02 2018.

[2] X. Lei, Z. Zhang, and P. Dong, "Dynamic path planning of unknown environment based on deep reinforcement learning," *Journal of Robotics*, vol. 2018, pp. 1–10, 09 2018.

[3] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. MIT Press, 2017.

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

[5] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 09 2015.

[6] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[7] L. Amado and F. Meneguzzi, "Q-table compression for reinforcement learning," *The Knowledge Engineering Review*, vol. 33, 01 2018.

[8] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674–690, 1997.

[9] L. ji Lin, "Reinforcement learning for robots using neural networks," 1992.

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

[11] S. Mousavi, M. Schukat, and E. Howley, “Deep reinforcement learning: An overview,” 06 2018, pp. 426–440.

[12] J. Munk, J. Kober, and R. Babuska, “Learning state representation for deep actor-critic control,” 12 2016.

[13] S. Lange and M. Riedmiller, “Deep auto-encoder neural networks in reinforcement learning,” in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 2010, pp. 1–8.

[14] D. Kimura, “Daqn: Deep auto-encoder and q-network,” 06 2018.

[15] A. A. E. T. D. Deepak Pathak, Pulkit Agrawal, “Curiosity-driven exploration by self-supervised prediction,” 2017.

[16] J. Oh, S. Singh, and H. Lee, “Value prediction network,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 6118–6128. [Online]. Available: http://papers.nips.cc/paper/7192-value-prediction-network.pdf

[17] A. Savkin and C. Wang, “Seeking a path through the crowd: Robot navigation in unknown dynamic environments with moving obstacles based on an integrated environment representation,” *Robotics and Autonomous Systems*, vol. 62, 10 2014.

[18] F. Pianosi, K. Beven, J. Freer, J. W. Hall, J. Rougier, D. B. Stephenson, and T. Wagener, “Sensitivity analysis of environmental models: A systematic review with practical workflow,” *Environmental Modelling Software*, vol. 79, pp. 214 – 232, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364815216300287

[19] Y. Zhu, R. Mottaghi, E. Kolve, J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, “Target-driven visual navigation in indoor scenes using deep reinforcement learning,” 05 2017, pp. 3357–3364.

[20] V. Bulitko and G. Lee, “Learning in real-time search: A unifying framework. j,” *J. Artif. Intell. Res. (JAIR)*, vol. 25, pp. 119–157, 01 2006.

[21] J. Zeng, R. Ju, L. Qin, Y. Hu, Q. Yin, and C. Hu, “Navigation in unknown dynamic environments based on deep reinforcement learning,” *Sensors*, vol. 19, p. 3837, 09 2019.

[22] J. T. L. L. M. B. W. Zhelo, O.; Zhang, “Curiosity-driven exploration for mapless navigation with deep reinforcement learning,” 2018.

[23] H. Bae, K. Gidong, J. Kim, D. Qian, and S. Lee, “Multi-robot path planning method using reinforcement learning,” *Applied Sciences*, vol. 9, p. 3057, 07 2019.

[24] G. Yen and T. Hickey, “Reinforcement learning algorithms for robotic navigation in dynamic environments,” vol. 2, 02 2002, pp. 1444 – 1449.

[25] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, “Learning to navigate in complex environments,” 11 2016.

[26] H. Van Hasselt, “Double q-learning,” 01 2010, pp. 2613–2621.

[27] P. Mannion, S. Devlin, K. Mason, J. Duggan, and E. Howley, “Policy invariance under reward transformations for multi-objective reinforcement learning,” *Neurocomputing*, vol. 263, pp. 60–73, 11 2017.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013, cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[29] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. P. Singh, “Action-conditional video prediction using deep networks in atari games,” *CoRR*, vol. abs/1507.08750, 2015. [Online]. Available: http://arxiv.org/abs/1507.08750

[30] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, “Integrating state representation learning into deep reinforcement learning,” *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1394–1401, 2018.

[31] N. Botteghi, R. Obbink, D. Geijs, M. Poel, B. Sirmacek, C. Brune, A. Mersha, and S. Stramigioli, “Low dimensional state representation learning with reward-shaped priors,” 07 2020.

[32] E. Rohmer, S. P. N. Singh, and M. Freese, “Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework,” in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[33] S. James, M. Freese, and A. J. Davison, “Pyrep: Bringing v-rep to deep robot learning,” *arXiv preprint arXiv:1906.11176*, 2019.

[34] H. Van Hasselt, “Double q-learning.” 01 2010, pp. 2613–2621.

Appendix A

Double Q-Learning

As introduced in the Chapter 1, the deep Q-network added a target Q-network based on former Q-learning algorithms, which improved the instability problem. However, this update mode of the Q-learning algorithm leads to an overestimation problem of the action values, which estimates the value of a state too optimistically. Then Double Q-network algorithm is proposed to optimize this problem [5]. The DDQN algorithm evaluates the greedy policy by maximizing the value of the online network and estimates the value using the target network, instead of selecting and estimating an action with the maximum value of the target network.

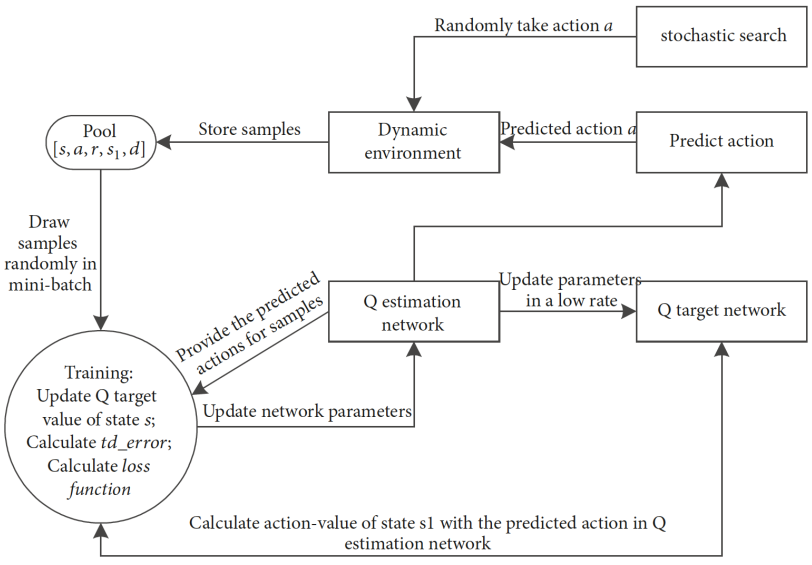


Figure A.1: The framework illustration of DDQN [2]

The target value y_i is calculated as:

$$y_i = E\left[(r + \gamma Q(s', argmax(Q(s', a'; \theta_i)); \theta_i') \mid s, a\right], \tag{A.1}$$

The loss function is:

$$L_i(\theta_i) = E\left[(r + \gamma Q(s', argmax(Q(s', a'; \theta_i)); \theta_i') - Q(s, a; \theta_i))^2\right] \quad \text{(A.2)}$$

The DDQN algorithm is described in Algorithm 3. The framework of the DDQN algorithm is shown in Figure A.1.

Algorithm 3 Double Q-learning(Hasselt et al. [34])

Require: Initialize primary network Q_1, target network Q_2, $\forall s \in \boldsymbol{S}, a \in \boldsymbol{A}(s)$, $\tau << 1$

for each iteration **do**
Initialize S
for each environment step **do**
Observe state s_t, reward r_t, and select $a_t \sim \pi(a_t, s_t)$ derived from Q_1 and Q_2
Execute a_t and observe next state s_{t+1} and reward $r_t = R(a_t, s_t)$
With 0.5 possibility:
$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + \gamma Q_2(s_{t+1}, argmax_a Q_1(s_{t+1}, a) - Q_1(s_t, a_t))$
else:
$Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_t + \gamma Q_1(s_{t+1}, argmax_a Q_2(s_{t+1}, a) - Q_2(s_t, a_t))$
Update states: $\boldsymbol{S} \leftarrow \boldsymbol{S}'$