NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

Unmanned Ground Vehicle Indoor Navigation Based on Deep Reinforcement Learning

YUECI DENG

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

2019

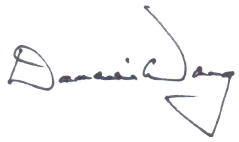
Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

[Input Date Here]	[Input Signature Here]
20, May, 2019	Deng Guoci
.....	.....
Date	[Input Name Here]

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.



[Input Date Here]

23 May 2019

Date

[Input Supervisor Signature Here]

Wang Dan Wei

[Input Supervisor Name Here]

Authorship Attribution Statement

This thesis **does not** contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

[Input Date Here]	[Input Signature Here]
20, May, 2019	Deng Gueci
Date	[Input Name Here]

Contents

Abstract

This dissertation aims to provide the methods of using Deep Reinforcement learning algorithm to train the UGV in simulation such that the trained UGV can reach a random target position and avoid the obstacles without any prior knowledge and model of environment. First, the basis of reinforcement learning, deep learning and deep reinforcement learning is introduced in chapter 2. In chapter 3, the the detail approaches used in this dissertation are described, including the software tools and algorithms that are used to build the simulation environment for training. We use three advanced and prevalent deep reinforcement learning algorithms to solve the expected tasks and design novel reward functions to increase the convergent capability. The Whole objective is divided into three steps, and the implementation process is included in chapter 4, where the main results are shown. The technical discussion and analysis about the problems of training the reinforcement learning system are included in chapter 5. Finally, the conclusions and recommendation to the future works are presented in chapter 6.

Keywords: UGV, target reaching, obstacles avoidance, deep reinforcement learning, reward function.

Acronyms

UGV Unmanned Ground Vehicle

ML Machine Learning

SL Supervised Learning

USL Unsupervised Learning

RL Reinforcement Learning

DL Deep learning

DRL Deep Reinforcement Learning

DQN Deep Q-Network

MDP Markov Decision Process

ANN Artificial Neural Networks

DNN Deep Neural Network

DDPG Deep Deterministic Policy Gradient

TD3 Twin Delayed Deep Deterministic Policy Gradient

SGD Stochastic Gradient Descent

BN Batch normalization

ROS Robot Operation System

Symbols

s	state
a	action
S	set of all non-terminal states
$A(s)$	set of actions possible in state s
t	discrete time step
T	final time step of episode
S_t	state at t
A_t	action at t
R_t	reward at t
G_t	return (cumulative discounted reward) following t
π	policy
$\pi(s)$	action taken in state s under deterministic policy π
$\pi(a\|s)$	probability of taking action a in state s under stochastic policy π
$p(s'\|s,a)$	probability of transition from state s to state s' under action a
$r(s,a,s')$	expected immediate reward on transition from s to s' under action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy

$q_\pi(s,a)$	value of taking action a in state s under policy π
$q_*(s,a)$	value of taking action a in state s under the optimal policy
γ	discount-rate parameter
ε	probability of random action in ε-greedy policy

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Unmanned Ground Vehicle (UGV) is a mobile vehicle that can move without human real-time operation. Self driving cars are the most common application of UGV for commercial use. The advancement of scanning technologies such as Lidar, radar and machine learning with the aid of cloud computing and industrial internet of things have allowed the concepts of self driving cars and UGV to begin to see mainstream usage and adoption in reality.

Apart from public transportation, there are other application scenarios, that may be arguably important such as farms, construction sites, industrial complexes, factory floors, warehouses, and airports. UGV applications are particularly suitable for replacing so called dirty, dull and dangerous work in controlled environments like these because the infrastructure setup and cost, if any at all, can be kept to an acceptable minimum as compared to self driving cars to be used on public high ways. For example, in a warehouse management setup, the warehouse floor can be autonomously mapped by UGVs as they traverse their surrounding. In addition, well defined markers could be placed at points of interests to assist the UGV in decision making. Finally, electronic beacons, radio based or light based, could also be dispersed strategically allover perimeters for further navigation assistance and decision making.

The core of developing a UGV system is the design of perception and motion planning system [1]. Perception system allows the UGV to obtain environment information around it, such as spatial position, visual information like image from the front side camera, or 3D information from Lidar. The motion planning system is a decision maker that uses the information from perception system and generates the optimal actions to achieve the goal. A typical UGV system components structure is illustrated in Figure 1.1.

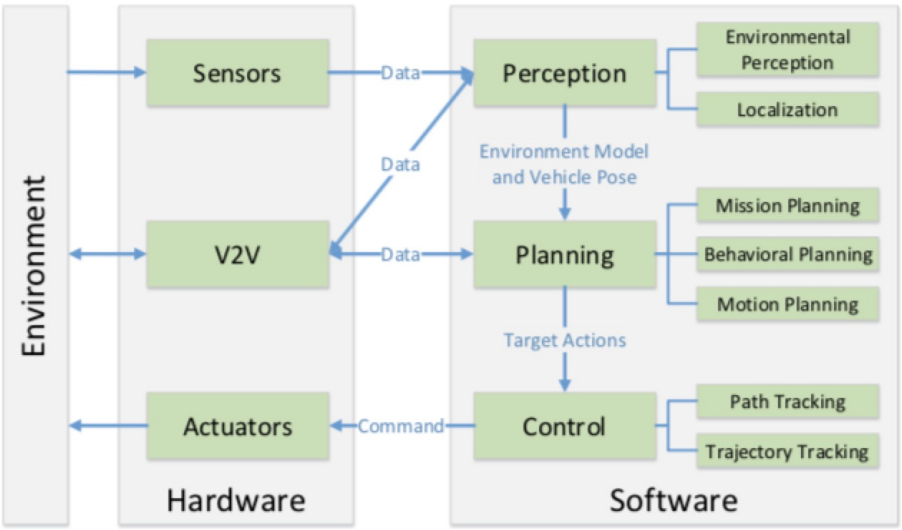


Figure 1.1: UGV system overview [1].

Indoor navigation aims to program the mobile vehicle such that it can move to the desired target position from the initial position without any collision with obstacle. Conventional methods, such as simultaneous localization and mapping (SLAM), need a prior model of the environment with observation of dense laser signals. There are two drawbacks for this method: (1) the time-consuming building and updating of the environment model and (2) fail to deal with dynamics around the environment. To make the UGV intelligent and handle difficult and complex problems, the more advanced algorithm should be researched and developed with robust and low cost characteristic.

1.2 Motivation

In recent years, with the dramatic success in deep learning (DL), the deep reinforcement learning (DRL) method is proposed and has achieved inspiring results in video games and simulation control agents. There are some works that use DRL method to solve the robotic problem like manipulation and mobile robot with continuous control [5] [6] [7] [8].

Unlike conventional methods that need model of the working environment of the robot, the DRL methods can provide model-free training and have good generalized characteristic. Besides, the robot can be trained in simulation and then be applied to real implementation without large cost in time. Navigation is a typical application scenario in mobile robot and UGV, and is suitable to solve by using DRL methods, which save the cost of rebuilding of the environment model and can deal with dynamic problem.

1.3 Objectives and Specifications

The objectives of this research project is to train the UGV using deep reinforcement learning in simulation environment such that the UGV can learn a good policy to reach a random target position while avoiding the obstacle. The objectives is divided into three steps as follow:

- **Task 1 - obstacle avoidance:** we buid a simple square maze in simulation. The UGV should travel around the maze without any collision after being trained by deep reinforcement learning algorithm.
- **Task 2 - random target position reaching:** we build an empty world and generate the target randomly. The UGV should navigate to the target using the spatial information.
- **Task 3 - Target reaching and obstacle avoidance:** we build a simple indoor environment with obstacle and random target generation. The UGV should achieve the goals described above after training.

1.4 Major contributions of the Dissertation

Three contributions are achieved in this research project, which are as follow:

- Implementation of Deep Reinforcement Learning algorithm (DRL) in Robot Operation System (ROS) system using Pytorch. Three different algorithms are implemented, including Deep Q-network (DQN), Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic Policy Gradient (TD3). The algorithms are written with Pytorch in Python.

- Build the required environment models in simulation and train the UGV to achieve the objectives.

- Design novel reward functions to accelerate the convergent speed. For three different tasks, the reward function is designed to suit the environment and DRL algorithms.

1.5 Organization of the Dissertation

The subsequent chapters of this dissertation are organized as follow. Chapter 2 provides a literature review on basis of reinforcement learning, deep learning and deep reinforcement learning, in which the fundamental theorem and important concepts and equations are presented and explained. Chapter 3 describes the methods, models and algorithms that are used and designed in this project in detail. Chapter 4 shows the implementation of the experiment and their results of all the tasks. Chapter 5 and chapter 6 give the technical discussion, the conclusion and suggestions for future works, respectively.

Chapter 2

Literature Review

This chapter describes the basic theorem of reinforcement learning (RL), including its objectives, important elements and methodology. Besides, the concepts of deep neural network (DNN) and its optimization method are introduced. The popular deep reinforcement learning (DRL) algorithms will be described after these two topics, in which the algorithm utilized in this research project will be discussed in detail.

2.1 Basis of Reinforcement Learning

Reinforcement learning is a sub-domain of machine learning (ML). However, Unlike typical machine learning algorithm, such as supervised learning (SL) and unsupervised learning (USL), which aim to fit a function to achieve classification or regression with or without labeled data, the objectives of reinforcement learning is to teach a learnable agent so that it can interact with external environment to achieve a specific goal [2].

There are two groups of scientists that divide the reinforcement learning into two categories. One of them thinks that the agent should be taught using the experience from human, which is also called imitation learning [9]. Another group believes that the agent must learn from its own experience. Experience from other kinds of agent or system are meaningless because different agents have different internal specification and different observation from the environment. In this dissertation, our focus is the second group and all the included theorems are come from it.

Reinforcement learning is quite different from supervised learning and unsupervised learning, which use training data from various of sources. Another difference is that, the data used in reinforcement learning algorithm is almost time-dependent or can be seen as time series, while data in SL and USL is actually time-independent. Therefore, we can know that the RL system process a sequence of data obtained from the environment and generate a sequence of corresponding output from the learning agent to reach the expected goal. The detailed description will be included in next subsection.

There are some typical RL applications which have shown remarkable achievements in recent years.

- **Vdieo Game:** [3] [10] show that using deep learning and reinforcement learning can teach the computer to learn how to play video games and even beat human in some aspects. The agent (computer) can directly take raw video frames as input and generate command to control the action in the game.
- **The board game GO:** Google DeepMind has researched and developed a super computer program, AlphaGo [11] using Monte Karlo search tree [2] and Deep learning to play the GO game and beat greatest Go game player in the world.
- **Robotics:** [12] [13] [5] [6]successfully apply deep reinforcement learning (DRL) in mobile robot indoor navigation. [8] [14] show that the intelligence motion planning for robotic manipulation can be implemented using DRL with both vision input.
- **Recommendation System:**This application has been developed for many years in IT companies and in the current ten years it is becoming increased mature in news, video and music recommendation [15] [16] [17].

2.1.1 Elements of Reinforcement Learning

The well defined reinforcement learning system is illustrated in Figure 2.1. To mathematically model the system, the four elements should be identified: the policy, the

reward function, the value function and the model of environment, which including the state and action.

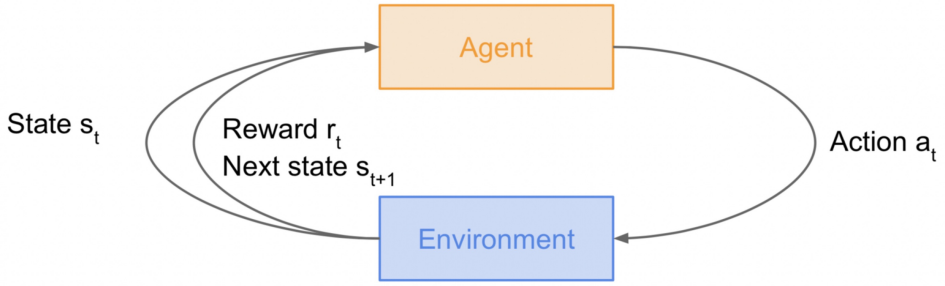


Figure 2.1: Reinforcement learning system. Source: Lecture 14: Deep Reinforcement learning, Stanford CS231n, http://cs231n.stanford.edu/2018/syllabus

The policy is a function that maps the state input to action output, and it can be treated as a part in the agent. The form of the policy function is typically a look-up table that stores a value for each state-action pair, which can be described as tabular method. Recently, some research find that using the neural network to be a function approximator dramatically increases the performance of the reinforcement learning [3] [10]. The policy is most crucial in RL system because it is directly related to the behaviour of the agent.

The reward function implicitly defines the goal of the agent in the given environment. According to Figure 2.1, in each time step, when the agent generates an action, the environment will give an immediate evaluation signal back to the agent. Now comes the simple mathematical description of the objectives in RL system, that is to maximize the cumulative reward over a finite or infinite time sequence. Intuitively, the reward function evaluates the action of current state, but in most of time, the reward depends on not only the action itself, but also the current state, the next state or even other important factors in the RL system. Sometimes, different RL systems with different goals or tasks have completely different factors that matter. The reward signal is the basis for policy learning and the policy will tend to select the action with high reward.

Different from reward function which gives an instant signal, the value function defines the total reward an agent can expect to obtain in the future, with beginning at the

current state. Thus, it is the value of state, or in some cases called state-action pair. The value of the state implicitly shows what is good in the future time sequence. For example, in each time step, the agent will take an action and enter a new state with an immediate reward. Sometimes the value of the reward is low, but the value of this state-action pair is high, indicating that the following actions will receive high instant reward and the cumulative amount will be maximum.

By intuition, the reward function is most directly related to the performance of RL algorithm. However, the value function is most concerned for agent decision making, or says, the adjustment of policy [2]. The core of reinforcement learning, is the method to estimate the value and hence find the optimal policy that can maximize the total reward in the whole process. More detailed description about policy, reward function and value function is in Section 2.2 and 2.3.

The final element in RL system is the model of environment, or more specific, the action space and state space. The reason why we call them space is that both action and state in a typical RL system are not comprised by only one element. For example, for the robotic manipulation system, the action is the motor control of several joints and the state could be a concatenated information, such as spacial position of joint, joint angle and velocity of joint. The definition of action and state space is also important and should be obey the Markov property [2], which is discussed in the following subsection. The basic and simple principle to design the state and action space is that, they should be reasonable. For action, it should be set within a limited boundary to avoid intense oscillation and be allowed enough dynamic movement. For state, it should be informative and fully contain the critical things in the given environment. In different RL system the definition of action and state can be completely different, and usually the goal of the RL system plays an important role in design of action and state space.

2.1.2 Exploration and Exploitation

The trade-off between exploration and exploitation makes reinforcement learning different from other learning method. Exploitation means that the agent must select the

action based on what it has already known. The policy actually acts as the brain of the agent and allow it to exploit in the environment to achieve the goal. Exploration lets the agent discover the state-action pairs as much as possible so that it can learn from them and find the optimal combination of state-action pairs. Both of them should be tried with a proper percentage and neither exploration nor exploitation can take the dominant part in training the RL system.

The most common method for exploitation and exploration is called ε-greedy action selection [2]. ε is the pre-defined threshold value, which is usually 0.1 or 0.01. In each time step, a random number will be generated with uniform distribution in the range of 0 to 1. if the random number is larger than ε, then the agent will select the action with largest value under the current state and this is why this method is so called greedy. If the random number is smaller than ε, the agent will start to select one action from the action space with equal probability for exploration. This method has been widely used for many years and is still used in some very famous reinforcement learning algorithm, such as Q-learning [2] [10] [3].

2.1.3 Returns

Based on the introduction above, the objectives of an agent in a RL system is to maximize the cumulative reward in the future. Formally, the cumulative reward, in this case, called expected return, is defined as

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T, \tag{2.1}$$

where T is the final time step. A sequence of time steps is called episode. Each episode ends when the agent reaches the terminal state, then the agent is reset to the initial state or to a sample from a distribution of initial state. In practice, the discount factor γ is usually introduced to the computation of return,

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{2.2}$$

where $0 \leq \gamma \leq 1$ is the discount factor. The discount factor results in the decay of future reward signal and force the return to a bounded value, especially when T is extremely large or even infinite.

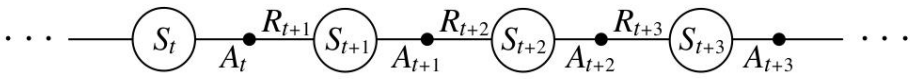


Figure 2.2: Reinforcement learning sequence [2].

2.1.4 Markov Property and Markov Decision Process

In the reinforcement learning system, the environment and the state signal should have the Markov property. Considering the environment respond to the action at time step t, it should be expected that the respond depends on all the state-action pairs happened before. Formally, the dynamics can be defined by a probability distribution,

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\}. \tag{2.3}$$

If the state has Markov property, then the respond of the environment depends on only the current state and action, leading to the simplified version of probability distribution,

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\}. \tag{2.4}$$

The environment and the goal are said to have the Markov property if (2.3) and (2.4) are equal. Besides, by iterating this relation, the future expected states and rewards can be predicted with the knowledge of transition equation of all possible state S and action A.

Although in some cases, the state signal is non-Markov, it is also reasonable to be seen as an approximation of the Markov state and used in a RL system. In short, the Markov property assumes that the future state depends on only the current state, and hence, the representation of the state signal must be informative. Otherwise, the policy learned from the experience of agent will be bad and can not perform well in the environment.

A reinforcement learning problem that satisfies the Markov property is called the Markov decision process (MDP). If the action and state spaces are finite, then it is called the finite Markov decision process. The complete MDP is defined by its state S, action A and its transition equation of the environment, which is

$$p(s^{'}|s,a) = \mathbf{Pr}\{S_{t+1} = s^{'}|S_t = s, A_t = a\}, \tag{2.5}$$

where $p(s^{'}|s,a)$ is called transition probability. Given the current state and action, the expected value of next reward is

$$r(s,a,s^{'}) = \mathbf{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s^{'}]. \tag{2.6}$$

These two quantities define the most important dynamics in MDP.

2.1.5 Value Function

The value function is the estimation of future cumulative reward that can be expected given the current state or current state-action pair. According to previous definition of policy, π, which is a function that maps each state, $s \in S$, to an action, $a \in A(s)$. The probability of taking action a when in state s is $\pi(a|s)$. Therefore, the value of a state s under the policy π is actually the expected return with starting in state s and can be defined as

$$v_\pi(s) = \mathbf{E}_\pi[G_t|S_t = s] = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \tag{2.7}$$

where $v_\pi(s)$ is called state-value function for policy π. what should be mentioned is that the value of terminal state is always zero. For state-action pair, the value is defined as

$$q_\pi(s,a) = \mathbf{E}_\pi[G_t|S_t = s, A_t = a] = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a], \tag{2.8}$$

where $q_\pi(s,a)$ is action-value function for policy π.

A fundamental theorem of value function is that it satisfies a recursive relationship, which is also called Bellman equation [2]. For any policy π and state s, the relation-

ship between the value of current state and next state is

$$\begin{aligned} v_\pi(s) &= \mathbf{E}_\pi[G_t|S_t = s] \\ &= \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ &= \mathbf{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,a') + \gamma \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,a') + \gamma v_\pi(s')] \end{aligned} \tag{2.9}$$

The solution of the Bellman equation is the value function $v_\pi(s)$. In most RL problem, the fundamental problem is to learn $v_\pi(s)$ from the Bellman equation, and hence find the policy that maximizes the long turn reward, which is called the optimal policy. Formally, π_* is the optimal policy if and only if $v_{\pi_*}(s) \geq v_{\pi'}(s)$ for all $s \in S$. The optimal policy is not necessarily unique, but all of them have the same optimal state-value function and optimal action-value function, which are

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s), \\ q_*(s,a) &= \max_\pi q_\pi(s,a), \end{aligned} \tag{2.10}$$

for all $s \in S$. Combined with equation (2.9), the relationship between successive optimal state-value function and action-value function are

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma(v_*(s'))] \\ q_*(s,a) &= \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \max_{a'} q_*(s',a')] \end{aligned} \tag{2.11}$$

Now that all the elements of reinforcement learning problem have been defined and described, the next step is to obtain the optimal policy, where the computation might be extensive. A problem is that the build up of the approximation of optimal value function is usually memory consuming. The reason is that, the value of each state-action pair should be stored in a table or matrix so that it can be updated during the

process of continuous calculation. This method is called tabular method. However, it is only suitable for small state and action space RL model. If the state and action space become extremely large or continuous, the number of state-action pair becomes infinite and can not be fully stored in computer memory. Therefore, the neural network was introduced into reinforcement learning and used for approximation of value function. This method is called Deep Reinforcement learning (DRL). The concept 'Deep' means that the deep neural network (DNN) or usually deep convolutional neural network (CNN) have more than one hidden layer and can be extended to a very deep structure, which can be utilized as the policy in RL system. DRL has been proved to solve a number of difficult problems with good performance. Besides, by using CNN, raw image frames are allowed to used as state in RL problem, leading to a more direct definition of state and reducing the complexity of extracting useful information from the environment.

2.2 Tabular Methods

In this section, some traditional methods used to solve the reinforcement learning problem is discussed. The word 'tabular' indicates that the approximation of the optimal policy is represented by a table, which stores each possible state-action pairs and its value obtained from the value function. This kind of method is quite useful for some RL systems that have small state and action space. For exploitation, the action is selected based on the largest value of state-action pair. For exploration, the action is selected randomly based on uniform distribution. The condition for balancing the exploitation and exploration is a value called ε, and this is why the policy is called ε-greedy.

There are three typical methods to solve the value function, including dynamics programming, Monte Carlo methods and temporal-difference (TD) learning [2]. Each of them has its advantages and disadvantages. Dynamic programming has solid mathematically theorem but requires a full knowledge of the environment, like the transition function of the state. Monte Carlo methods do not need the environment model but

is not suitable for recursive computation. Finally, temporal-difference learning overcomes the two drawbacks of above methods, but it is more complex and sometimes unstable. The detail of Monte Carlo methods is not discussed in this dissertation because its theorem was not used in this research project. We will focus on the value iteration method, a algorithm from dynamic programming, the Q-learning and Sarsa, which are from TD methods.

2.2.1 Value Iteration

The value function is defined in a sequence form, v_k, where k indicates that the value function is changed over the time and k is the time sequence. Recalling that the recursive equation 2.11 of value function shows the relationship of successive state of value function, now it can be rewritten as

$$v_{k+1}(s) = \max_{a \in A(s)} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma(v_k(s')]. \tag{2.12}$$

if we repeat the computation of this recursive equation infinitely, it can be proved that v_k will converge to the optimal value function v_*. However, in practice, the computation is terminated under some conditions, in which v_k can be seen as the approximation of v_*. The detailed algorithm is below.

Algorithm 1 Value iteration

Initialize θ for termination
Randomize $V(s)$ for all $s \in S$

repeat
  $\Delta \leftarrow 0$
  for each $s \in S$: **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
Output a deterministic policy π, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

According to the algorithm above, to obtain the expected policy, the transition func-

tion of the state-action pairs must be known, then the final result is easily to computed. Otherwise, the policy can not be computed if the model of the environment is unknown.

2.2.2 Temporal-Difference Learning

TD method uses the experience of agent to solve the solution of optimal value function, which allows it perform the estimation under the unknown environment. For tabular methods, the policy is usually deterministic, and the experience is collected following the policy π. Therefore, the simplest TD method, called $TD(0)$, uses the difference between the expected return and the value of current state to update the current estimation of $V(s)$,

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.13}$$

The biggest difference between value iteration and TD learning is that, TD learning does not requires the model of environment and it finds the optimal value function only based on the current and next time step value of state-action pairs. Therefore, the problem of exploitation and exploration is important in TD methods. For exploitation, the ε-greedy policy selects the action with maximum value of state-action pair. For exploration, the agent selects the action from the action space with equal probability. In each time step, the agent takes an action A_t based on the current policy, which decides whether to exploitation or exploration. Then it will receive the instant reward R_{t+1} and enter the next state S_{t+1}. The update can be done by using these values and this process is repeated until the agent reaches terminal state. TD learning is of vital importance in reinforcement learning history, and it is also the basis of the following most significant algorithms, Q-learning and Sarsa.

2.2.3 Sarsa and Q-learning

The full name of Sarsa is state-action-reward-state-action, where the first state-action means the value of state-action pair in current time step, and the second is the value of next time step. Sarsa is an on-policy algorithm. Recalling that the equation 2.13 shows the update of value function during training, Sarsa learns the state-action function and

the update equation is

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)]. \tag{2.14}$$

The reason why it is called on-policy is that, the value of state-action pair in next time step is obtained under the current policy. The update of action-value function is executed after every transition from a non-terminal state. If the state S_{t+1} is terminal, then the value of the corresponding state-action pair is set to zero.

The most significant breakthrough in reinforcement learning was the off-policy TD algorithm, Q-learning [2]. The simplest one-step Q-learning is defined as

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)]. \tag{2.15}$$

The difference between on-policy algorithm Sarsa and off-policy algorithm Q-learning is that, Q-learning learns the action-value function directly using the approximation of optimal action-value function q_* and is independent to the current policy. While Sarsa uses the current policy for temporal difference calculation. The small change in Q-learning leads to a dramatic improvement in convergent speed of the policy estimation. The detailed algorithms of Sarsa and Q-learning are shown below.

Algorithm 2 Sarsa: the on-policy TD algorithm

Set $Q(terminal-state,\cdot) = 0$
Randomize $Q(s,a)$ for all $s \in S$, $a \in A(s)$

repeat(for each episode)
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  repeat(for each step of episode)
    Take action A, observe R, $S^{'}$
    Choose $A^{'}$ from $S^{'}$ using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S^{'},A^{'}) - Q(S,A)]$
    $S \leftarrow S^{'}; A \leftarrow A^{'}$
  until S is terminal

TD learning methods brings a breakthrough in reinforcement learning and have been proved that they can converge to the optimal value function with enough iterations.

Algorithm 3 Q-learning: the off-policy TD algorithm

Set $Q(terminal-state,\cdot)=0$
Randomize $Q(s,a)$ for all $s\in S$, $a\in A(s)$

repeat(for each episode)
  Initialize S
  repeat(for each step of episode)
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, $S^{'}$
    $Q(S,A)\leftarrow Q(S,A)+\alpha[R+\gamma\max_a Q(S^{'},a)-Q(S,A)]$
    $S\leftarrow S^{'}$
  until S is terminal

However, in some problems with complex environment, where the state space and action space are extremely large or even infinite, tabular based algorithms are no longer feasible due to the limited memory and search efficiency. Although TD methods are not perfect, they form the basis of modern reinforcement learning theorem and are still important to study and develop.

In next section the deep reinforcement learning methods will be introduced. The basic concept and theorem of deep learning will be described first, then the most significant algorithm, Deep Q-learning (DQN) will be discussed in detail.

2.3 Deep Reinforcement Learning

According to previous discussion, the limitation of tabular methods in reinforcement learning is the constraint of state and action space, which narrows their application in real-world problems. In tabular methods, the value function is represented by a table, which stores every state-action pairs. During the TD learning process, the value of action-value function is obtained by searching the table and updated using the temporal difference. The larger the amount of state-action pairs, the longer the searching time will be spent. To overcome the state space constraint and save the time cost in searching, the deep neural network is utilized to represent the value function [10] [3] and is well known as Deep Q-leaning (DQN). This algorithm combines both deep learning model and Q-learning model and has been successfully utilized in some applications,

especially the video games.

In this section, the focus is the fundamental theory of deep learning and DQN algorithm mechanism. Although DQN is not used in this research project, its methodology is heuristic and useful in development of more advanced algorithm.

2.3.1 Artificial Neural Networks

The concept of Artificial Neural Networks (ANN) can be dated back to 1960s and was proposed to construct a mathematical representation for information processing in human biological system. The simple structure of ANN can be illustrated in Figure 2.3.

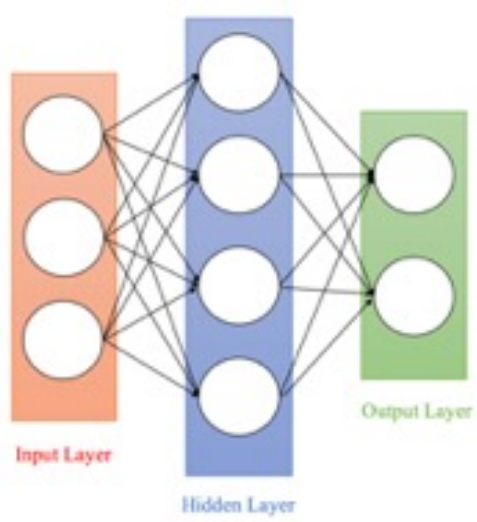


Figure 2.3: Typical ANN model with one hidden layer.

Basically, artificial neural networks, or simply called neural networks, is a widely used supervised learning algorithm in machine learning for classification tasks with non-linear feature distribution. The input layer loads the feature set extracted from the data. The hidden layer maps the input features to a more complex feature domain with highly non-linear characteristic. The output layer is a classifier that output the result of classification.

In supervised learning, training is the most critical step, because it enables the math-

ematical model to learn the characteristic of training dataset and hence the model can obtain 'knowledge' from such kind of data. The training dataset consists of data and their labels. The training step of ANN contains two parts, called feed-forward and back-propagation. In feed-forward step, the training data is sent to ANN and output the classification prediction. After that, the difference between the prediction and corresponding labels, known as loss, is used to update the weights of the model.

Feed-forward

Based on Figure 2.3, assuming that the input data is $x \in R^{i\times 1}$, the functional expression of hidden layer is $f_h(\theta)$ and output layer is $f_o(\theta)$, where i, h and o represent the number of neuron. When input data move from input layer to hidden layer, the result at hidden layer is

$$y = f_h(w_i x + b_i), \tag{2.16}$$

where w_i and b_i are the weight and bias between input layer and hidden layer. Then it moves to output layer and the output is

$$Y = f_o(w_h x + b_h), \tag{2.17}$$

where Y is a vector, which contains the values for each class. The decision of the final prediction is made based on the index of the maximum value in the output vector. The number of nerve cell in each layer is predefined by the designer and sometimes, the number hidden layer can be increased to more than one. $f_h(\theta)$ and $f_o(\theta)$ are called activation function and used to increase the nonlinear capacity of the model. These functions are crucial in ANN. In practice, the training data is not always linear distribution, and hence selecting an appropriate activation function not only enhances the capacity of the model but also saves the training time and hence reduces the computational complexity. For hidden layer, the most common activation function is called ReLU [18], which is defined as

$$y = \max(0, x). \tag{2.18}$$

For output layer, the activation function is usually defined as

$$y = \frac{e^{x_i}}{\sum_{k=1}^{N} e^{x_k}}, \tag{2.19}$$

where N is the total number of classes. This function is called Softmax [18], which returns the a normalized vector and its value can be seen as the class probability.

Back-propagation

The purpose of back-propagation is to optimize the weights so that the error between prediction and target can be reduced. The formula used for computing the error is called loss function. There are two common used loss functions in machine learning, one is called mean square error,

$$L = \sum_{i=1}^{n} (y - y^{'})^2, \tag{2.20}$$

where n is the amount of data used for loss calculation, and $y^{'}$ is the labels of the data. This loss function is used mainly in regression problem. For classification problem, it is no longer suitable, because the loss function must be convex so that the optimal solution can be found by using gradient descent method. Since the output of classifier is more than one element, the mean square error can not generate the convex loss when the number of output element is multiple. Therefore, another famous loss function, cross-entropy, is used in classification, which is defined as

$$L = -\sum_{i=1}^{n} y^{'} log(y). \tag{2.21}$$

Now that the loss is computed, what should be done next is decreasing the loss. Gradient decent [18] is the most typical algorithm. By applying this algorithm, the weights in each layer can be computed by taking the derivate using chain rule. The basis form of the update rule is

$$w \leftarrow w - \eta \nabla L(w), \tag{2.22}$$

where η is learning rate or training step, which defines how much the weight is changed in each back-propagation step.

The combination of feed-forward and back-propagation forms a step in training the ANN model, and normally, to train a good model, the process need to be executed more than a million times. Thus, if the training dataset is very large, the computational complexity would be increased, resulting in the decrease of convergent speed. To accelerate the convergent speed, the stochastic gradient descent (SGD) [19] was proposed and currently becomes the standard optimization algorithm in deep learning research area. In gradient descent, the loss is computed by feed-forward all the training data, which is very time consuming when the amount of data is huge. By contrast, SGD randomly groups a small number of data into a mini batch from the whole dataset without replacement for loss computation, and use this loss to perform gradient descent. Therefore, the back-propagation is performed only using a batch of data and hence the duration between each steps is reduced dramatically. There are some improvements of SGD. The most remarkable one is Adam [20], which uses an adaptive learning rate and the momentum of gradient to update the model weights. This optimization algorithm is also used in this research project.

2.3.2 Deep learning

ANN forms the basis of deep learning, which receives remarkable achievement in many frontier domains, such as computer vision, natrual language processing and speech recognition. The most significant skill used in deep learning is two-dimension convolution implemented by using a moving small kernal, or called filter. The kernal share ites weight along the specific channel of the feature map, and is optimized using gradient descent based algorithm. By replacing the linear transform, the neural networks using convolution kernal is called convolutional neural networks, and there are three basic components, including convolution layer, pooling layer and fully connected layer.

The convolution layer extracts the features from input data. In deep learning, a typical

model usually contains a sequence of convolution layer, so that the features extracted from each layer can be mapped into a high dimension domain with high degree of non-linearity. This method is a distinguished improvement in machine learning. Traditionally, the features of data are designed by the experts, which have abundant experience on a specific domain, such as face recognition, iris recognition, speech recognition and so on. Now, the feature extractor becomes learnable and can be varied based on the characteristic of training data. A simple illustration of 2-dimension convolution operation is shown in Figure 2.4. After convolution, the activation function is also applied to the output before it is passed to next layer.

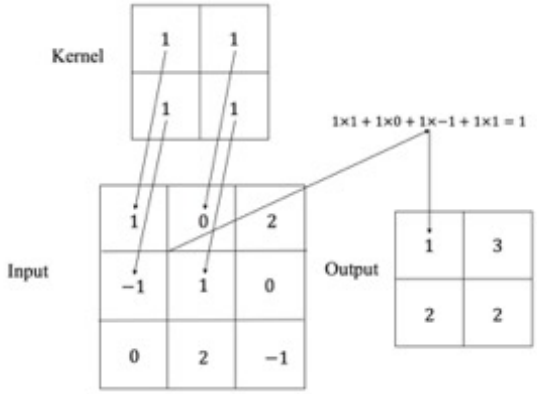


Figure 2.4: Convolution between kernal and input dara.

Pooling layer is often periodically inserted to the CNN model with the aim of progressively reducing the spatial size and hence reducing the amounts of parameters and computation [18]. It processes each depth slice of input independently with 'max' or 'average' operation. The typical size of kernel in this layer is 2×2 and stride is 1, resulting in the twice reduction of input size. Results find by researchers showed that max pooling performs better than average pooling in most of the cases. The illustration is below.

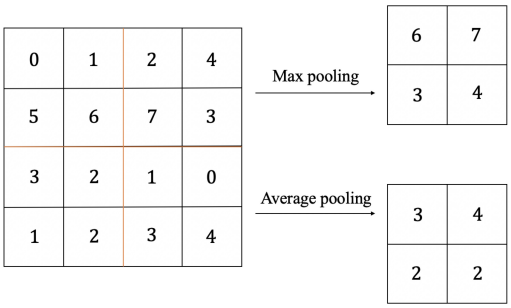


Figure 2.5: Max-pooling and average-pooling.

The fully connected layer is similar to the function of hidden layer in ANN, which performs linear transform of the extracted features. Normally, the result after transformation has the same number of elements with the number of class to be predicted. Then these transformed features are passed to the final output layer, a soft-max classifier. Figure 2.6 shows the example of image classification using CNN.

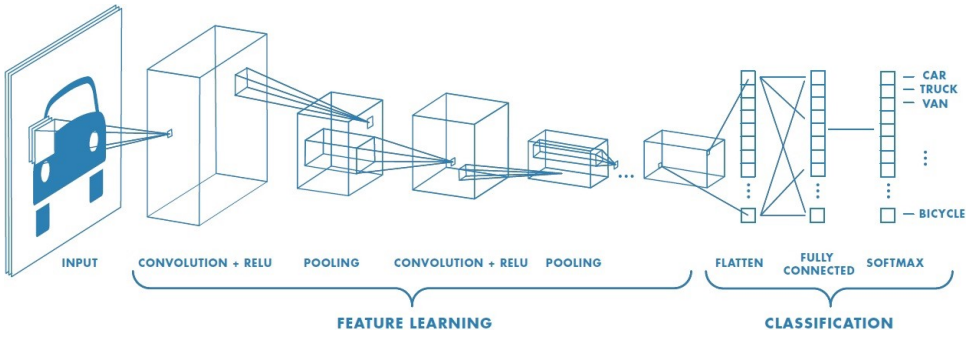


Figure 2.6: Image classification example using CNN. Source: Source : MathWorks, https://goo.gl/zondfq.

The structure of convolutional neural networks is flexible and can be designed based on the target-driven problem. For example, in image semantic segmentation domain, the fully connected layers is removed such that the final output can be a feature map that indicating a specific category [21]. This methodology is also applied in objection detection domain. Beside, recently, a mechanism called attention, is widely used in CNN to boost its feature extraction capacity by inserting some transformation layers between convolution layers [22] [23]. The invention of residual networks [24] leads a remarkable evolution in deep learning. Its solve a common and intractable problem in training the DL model, which is called gradient vanishment. This phenomenon

is caused by the propagation of gradient. When the model is very deep, the gradient would be very small after applying chain rule for derivation in a long run. Resnet saves this problem by dividing the layers into many sub-blocks and connecting the input to the output. Figure 2.7 shows the diagram of residual block.

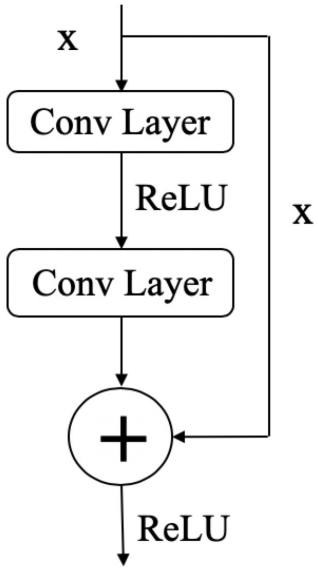


Figure 2.7: Resdual block.

2.3.3 Deep Q-Network

The advances in deep learning makes it possible to extract informative features from raw data without knowledge from experts. The mechanism of CNN model is quite suitable for function approximation in reinforcement learning domain. However, two existed issues makes RL problem is different from DL problem. The first is the delay between actions and the reward signal, which sometimes makes it inefficient compared with the association between input data and labels in supervised learning. The second is that the training data in deep learning is assumed to be independent, while in reinforcement learning the data is time correlative. Besides, the data distribution in RL problem is changed when the agent updates its value function and policy, which is problematic for deep learning that has the fixed data distribution.

To alleviate these problems, an mechanism called experience replay, which randomly

samples a batch of history transitions, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a memory $D = e_1, ..., e_N$. In each time step, the agent observes the current state and selects an action based on current policy and value function. Then it will receive the instant reward signal and enter next state. The four transition elements are grouped together and stored into the memory D.

The most critical part in Deep Q-learning is the method that updates the value function Q. In DQN algorithm, two Q functions are initialized, which are called evaluation Q function and target Q function, respectively. After storing the transition into memory, a batch of transitions is sampled randomly from the memory and sent to the two Q functions, which is represented by convolutional neural network or normal neural networks. For evaluation network, it computes the value of current state under current model weights. The target network computes the expected value of current value function $r + \gamma Q(s^{'}, a^{'})$,

$$Q(s,a) = E[r + \gamma \max_{a^{'}} Q(s^{'}, a^{'}) | s, a]. \tag{2.23}$$

Then the loss can be computed by these two values using mean square error. Here the reason why the loss function is MSE is that, finding the optimal Q function can be seen as the regression problem, and the target of DQN is to reduce the difference between evaluation function and target function. After obtaining the loss, the SGD algorithm is applied to optimize the weights of evaluation network. The weights of target network are not updated by gradient in every time step but updated by copying the weights from evaluation network. The number of time steps between each coping is set large to make sure that the difference between the two networks is large enough at the beginning of training. When the value function converges to the optimal value function, the difference will decrease. The full DQN algorithm is in algorithm 4.
The structure of Q function is flexible and can be defined using different layers with varied parameters. The network structure used in the famous paper [3] is shown in Figure 2.8.

Algorithm 4 Deep Q-learning with experience replay

Initialize replay memory D with capacity N
Initialize evaluation network $Q(\theta)$ and target network $Q(\theta^{'})$ with random weights
for episode=1,M **do**
  Initialize state
  for $t = 1, T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \max_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    $s_{t+1} \leftarrow s_t$
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Randomly sample mini-batch of transition (s_j, a_j, r_j, s_{j+1}) from D
    Set $y_j = r_j$ if s_{j+1} is terminal
    Set $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a^{'}; \theta^{'})$ if s_{j+1} is non-terminal
    Perform gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to $Q(\theta)$
    Every C steps set $Q(\theta^{'}) \leftarrow Q(\theta)$

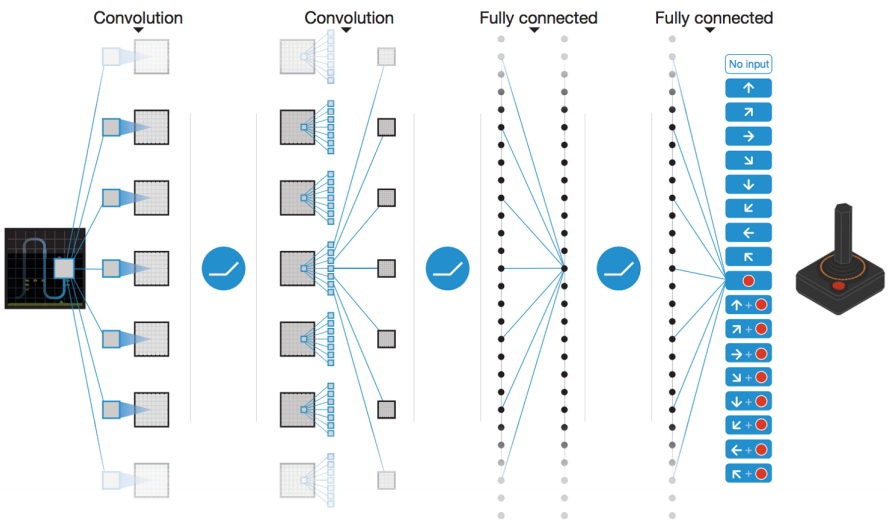


Figure 2.8: Structure of DQN [3].

The output of the network depends on the action space of the RL environment. DQN will give each actions a value given the current state, which is the Q value of state-action pair. The policy used to select the action is ε-greedy mentioned in previous section.

By using the experience replay mechanism, the correlation of all the transitions is broken and hence greatly improve the learning efficiency and solve the problem of dy-

namic data distribution. Since DQN uses its historical experience to update its weights, it is necessary to learn off-policy (recalling that the target Q value is generated using different model weights) and this is why the algorithm is called Deep Q Learning. The transitions in replay memory are sampled uniformly, which gives each transition the equal probability to be selected. This approach is limited because the memory does not consider the importance of the experience and usually overwrites with recent experience due to the finite memory size. This problem has been researched and attempted to tackle in [25].

The development of Deep Q Network leads to a dramatic improvement in reinforcement learning. DQN allows the agent trained in a more complex environment without sacrificing a lot of memory and saving the search time of value function. There are some advanced improvements on original DQN. [25] introduced a method that weights each historical experience and samples them with high priority. This method accelerates the convergent speed and avoid the problem of memory overwriting. [26] introduced the a double DQN algorithm, aims to solve the overestimation caused by ε-greedy policy. Research from [4] showed that for many states, it is unnecessary to estimate the value of each action because in these state, the action has no contribution on what is going to happen. To solve this problem, the Q network is modified to generate two path before finally computing the action value. One path is in charge of estimating the importance of state while another is for action. The modified network is called Dueling DQN and the comparison of original DQN is shown in Figure 2.9.

Although DQN has made great contribution in solving the reinforcement learning problem, it still has drawback and limitation in some special cases, especially in robotic control domain, where the action is not discrete but continuous. In next section, the policy gradient based RL method will be introduced to solve this problem. Actually, both DQN and gradient policy method are utilized in this research project and will be described in charpter 3 in detail.

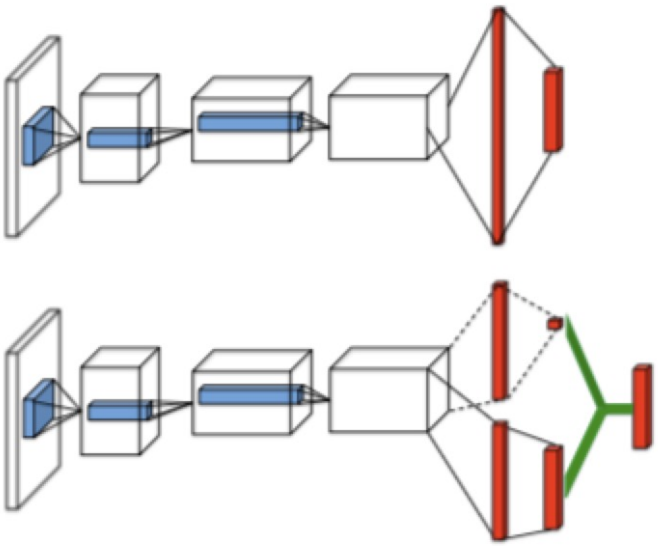


Figure 2.9: DQN and Dueling DQN structure [4].

2.3.4 Policy Gradient and Actor Critic

According to previous discussion, actions are selected based on the action-value function under the policy π, which can handle the problem with finite action space. When the action space is infinite, it is impossible for Q function to evaluate the value of all actions. Policy gradient based method [27] solve this problem by generating the continuous action directly using the policy. Therefore, different from value function approximation method that finds the optimal value function $v_*(s)$ or action-value function $q_*(s,a)$, the policy gradient method learns its policy directly and uses the gradient for policy updating. The policy is defined as

$$\pi_\theta(a|s) = p(a|s,\theta). \tag{2.24}$$

where θ is the parameters of policy, which can be a neural network or other function approximators.

Another difference is that, the policy using in value function based method is deterministic, like ε-greedy, which selects the action with largest value. However, the policy of gradient based method is stochastic, which means that the action selection depends

on the weights of policy and would be changed with optimization. A drawback of the policy method is that, the weights of policy are updated after each episode due to the loss function, which averages the score of each selected action. This mechanism is inefficient for convergence when the task is complicated. Thus, a novel algorithm called Actor-Critic, is proposed [2] combining both advantage of value function methods and policy gradient methods. The actor is the learnable stochastic policy updated using the value obtained from critic. The critic is the value function updated using TD error, like Q-learning or DQN. The action is chosen by actor while the critic generate a value as the estimation of this state-action pair. The architecture of this algorithm is below and comparison of these three different methods is shown in Table 2.1.

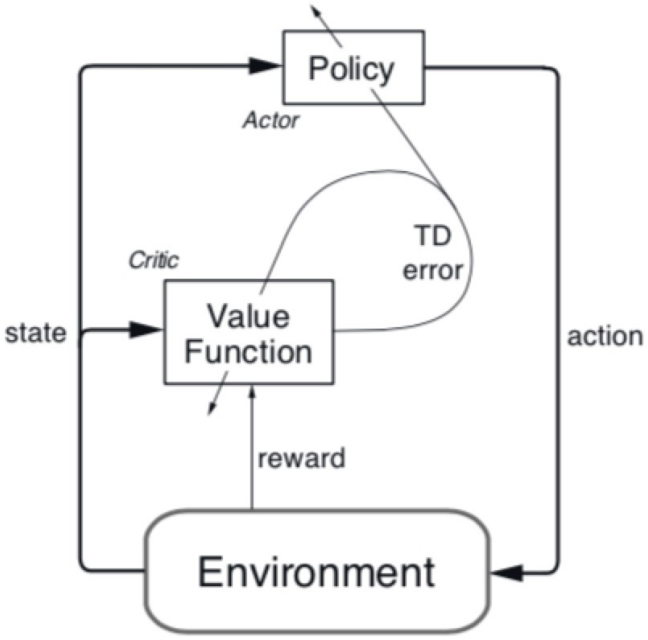


Figure 2.10: The architecture of actor-critic [2].

Table 2.1: Comparison of main methods in reinforcement learning

value function method	**policy gradient method**	**actor-critic**
learn value function	learn policy	learn both value function and policy
deterministic policy	stochastic policy	stochastic policy
discrete action space	continuous action space	continuous action space
update in each step	update in each episode	update in each step

According to the table above, the actor-critic method integrates the advantages from both value function and policy gradient method, and hence it achieves better perfor-

mance in complex tasks. There are several advanced algorithms that use the methodology from actor-critic and become the state-of-art in recent years. But here only the most significant one, Deep Deterministic Policy Gradient (DDPG) will be introduced in this paper, which is mainly used in this research project.

2.3.5 Deep Deterministic Policy Gradient

The invention of Deep Q network algorithm has successfully solved some complex reinforcement learning problems with unprocessed, high-dimensional sensory input. However, DQN can only handle discrete and low-dimensional action space. For physical control tasks, especially robotic related tasks, DQN cannot be directly applied due to the continuous action space. An obvious solution is to discretize the continuous action space. However, a limitation is that, the number of actions increases with the number of degrees of freedom. For example, for a N degree of freedom system with the discrete action range $a_i \in a_1, ..., a_k$, the total expected action number is k^N. The situation is worse for tasks that require smooth control with large range of discretization, resulting in an explosion of action space. Such large action spaces are difficult to explore efficiently, and thus successfully training DQN in this context is likely intractable. Additionally, naive discretization of action spaces throws away information about the structure of the action domain, which may be essential for solving many problems.

Actor-critic algorithm solves this problem while keeping the advantage of value function RL method. Motivated by the success of DQN, the Deep Deterministic Policy Gradient (DDPG) algorithm [28] was proposed using deep neural networks as function approximator that can learn the policy with high-dimensional sensory input and continuous action space. Besides, DDPG uses a recent advanced deep learning technique, batch normalization (BN) [29], which can accelerate the training speed of deep neural network by reducing the effect of covariant shift. The formula of BN layer is

$$y = \gamma \frac{x - E[x]}{\sqrt{Var[x] + \varepsilon}} + \beta, \tag{2.25}$$

where ε is a small offset that forces the denominator larger than zero. γ and β are the learnable parameters that shift the features in a fixed range.

DDPG is based on actor-critic structure. The critic is a DQN with both sensory state and selected action as input. The update of the Q network also uses TD error between evaluation and target networks computed by experience from replay memory . For the actor, an exploration policy is defined as

$$\pi(s_t) = \mu(s_t|\theta_t^{\mu}) + N, \tag{2.26}$$

where θ represents the weights of policy and N is the noise which can be chosen to suit the environment, such as uniform noise or some especial random processes. The algorithm of DDPG is listed below.

Algorithm 5 Deep Deterministic Policy Gradient

Initialize replay memory D
Initialize critic network $Q(s,a|\theta^{Q})$ and actor network $\mu(s|\theta^{\mu})$ with random weights
Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q^{'}} \leftarrow \theta^{Q}$, $\theta^{\mu^{'}} \leftarrow \theta^{\mu}$
for episode=1,M **do**
  Initialize a random process N for action exploration
  Initialize state s_1
  for $t = 1, T$ **do**
    Select action $a_t = \mu(s_t|\theta^{\mu}) + N_t$ based on the current policy and noise
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Randomly sample mini-batch of transition (s_i, a_i, r_i, s_{i+1}) from D
    Set $y_i = r_i + \gamma Q^{'}(s_{i+1}, \mu^{'}(s_{i+1}|\theta^{\mu^{'}})|\theta^{Q^{'}})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^{Q}))^2$
    Update the actor policy using the policy gradient:
    $\nabla_{\theta^{\mu}} J \approx -\frac{1}{N} Q(s_i, \mu(s_i|\theta^{\mu})|\theta^{Q})$
    update the target network:
    $\theta^{Q^{'}} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q^{'}}$
    $\theta^{\mu^{'}} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu^{'}}$

There are something new in DDPG algorithm. The first is that, the Q network (can be seen as value function) takes both state and action as input. This modification changes the state-value function $Q(s)$ into action-value function $Q(s,a)$. The actor is also a neural network with similar structure to critic and the action is generated in the output

layer using the formula

$$a = b \odot \tanh(y), \tag{2.27}$$

where 'tanh' is an activation function which forces the output within the range of $[-1,1]$. b is the action bound applied for each action degree of freedom, which is varied according to different environment.

Another difference is the balance between exploitation and exploration. In DQN, this is achieved using the ε-greedy policy, which assigns the probability to make decision on whether to exploit or explore. while DDPG uses a random process with relatively small maximum value for exploration and the deterministic policy for exploitation. The combination of them allows the agent to achieve both of them in each decision making step. In other word, DQN uses probability to balance the exploitation and exploration, while DDPG uses proportion. Definitely, the second method is more suitable for continuous control problems, which have the constraint of smooth and soft action control.

The last modification is the way of target network update. Reminding that DQN updates its target Q network by completely copying the weights from evaluation network every numbers of step. DDPG uses a soft update method, which is defined as

$$\theta^{'} \leftarrow \tau\theta + (1-\tau)\theta^{'}, \tag{2.28}$$

where $\tau \ll 1$. This means that the target weight are constrained to change slowly, and greatly improves the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which the robust solutions exists.

2.4 Summary

The basis principle of reinforcement learning has been introduced now, including the policy, value function, reward and environment model. Besides, the methodology of

deep learning is also introduced and the important algorithm, DQN and DDPG, are described in detail.

The primary problem of reinforcement learning is to find the optimal policy that maximize its cumulative reward in a long run by interacting with the environment. Whereas deep learning tries to learn a function that approximates a objective function using supervised learning. The combination of these two kinds of algorithm leads to a remarkable result in solving the complex tasks, and can be extended to the application of robotic control.

Chapter 3

Approach

In this chapter, the approaches adopted in this research project are mainly described, including the reinforcement learning algorithms, the design of environment and the design of reward function. Besides, the tools that realize these algorithms and environment models, are briefly introduced, such as the robot operating system (ROS), Gazebo (a simulation platform based on ROS) and Pytorch (a kinds of deep learning frameworks).

3.1 Software Tools

3.1.1 Robot Operating System

The robot operating system is not a standard operating system, like Linus, Windows or Mac. It is actually an integrated software frameworks for robotic software development. ROS has many useful application programming interfaces for different usage, including communications, sensing, navigation and action control. In addition, ROS provides a number of functional packages for fast algorithm development, such as 3D signal processing, image processing and dynamic control.

The program running on ROS should be written in a file called 'node'. Different ROS programs can communicate with each other through ROS topic and service. ROS topic connects the 'publisher' and 'subscriber', while ROS service is requested by the client.

These are two common and basis communication methods, and each of them has its own property. ROS topic is suitable for communication with high frequency and periodic response, while ROS service is usually used in pull-request applications and is more sensitive to the stable requirement.

3.1.2 Gazebo: a simulation platform

Gazebo is an environment simulation platform, which provides many physical models and their internal information. The user can define the model for robot and environment by using files with official format. By running ROS program on Gazebo, the robot can be controlled and interact with the simulation environment.

3.1.3 Pytorch: a deep learning frameworks

Pytorch is a popular deep learning frameworks and is widely used to build the deep learning models for both research and product. It supports distributed model training using CUDA, a parallel computing platform invented by Nvidia. Besides, it integrates a lot of prevalent functions from machine learning and deep learning and can be used from development to deployment.

The basis data structure in Pytorch is called tensor. A good mechanism is that the gradient of a tensor can be easily computed with by calling the built-in function, known as auto-gradient, by tracking all the operation applied on it.

Table 3.1 shows the main software environment used in this research project.

Table 3.1: Software Environment

Software	**Version**
Ubuntu	16.04
Python	2.7
Pytorch	1.0
ROS	Kinetic
Gazebo	7.0

3.2 Simulation Environment

Recalling that the objectives of this research project is random target position reaching in an environment with obstacle, the expected task is divided into three steps to achieve. At the first step, the obstacle avoidance is implemented using a double square maze. Then, the random target position reaching is implemented for the UGV with an empty environment. After successfully achieving this task, the obstacle avoidance is attempted to combine together and train the robot in a complex environment with obstacle. For all the environments, the physical simulations are constructed in Gazebo, while the communication between the UGV and environment as well as the specification are defined in software using Python scripts. The simulation UGV model is called 'husky', which is illustrated in Figure 3.1. The blue line is the laser beams, which have distance range $[0, 15]$ in meter and cover range $[-0.75\pi, 0.75\pi]$ in radian.

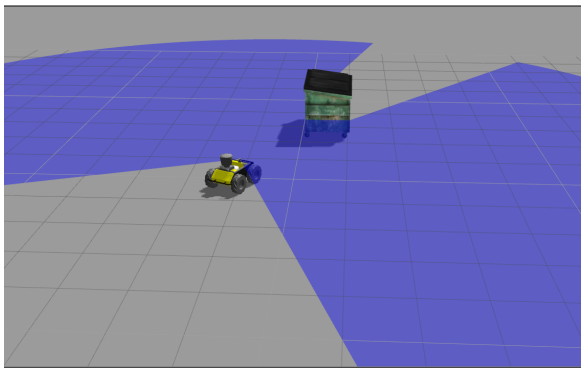


Figure 3.1: Husky simulation model with laser beams.

3.2.1 Obstacle Avoidance

The first task is obstacle avoidance. The robot is to travel around the double square maze and avoid collision with obstacle. Figure 3.2 shows the environment model, in which the robot moves inside the maze without any purpose but avoiding the contact of the wall. The cover range of laser is $[-\pi/4, \pi/4]$ now and the distance range is $[0, 5]$. The definition of state and action is listed in Table 3.2. The five laser beams are sampled from the raw signal with equal distance. The three different actions are defined with fixed value of linear and angular velocity.

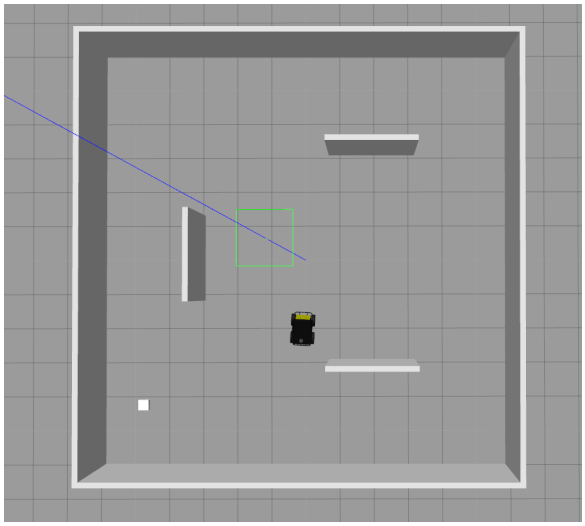


Figure 3.2: Double square maze for obstacle avoidance.

Table 3.2: State and action definition for obstacle avoidance.

Signal	Elements	Dimension
State	laser beams	$R^{1\times5}$
Action	straight, turn left, turn right	$R^{1\times3}$

3.2.2 Random Target Position Reaching

The environment used for target position reaching is an empty Gazebo world with only the husky model inside. The empty world is shown in Figure 3.3, where the white cube is just for visualizing the target position without physical collision. Besides, the target is randomly generating when the robot finishes an episode or reaches the terminal state. For this task, since the obstacle avoidance is not considered, the laser sensor signal is not used as the elements of environment state. Table 3.3 shows the definition of state and action.

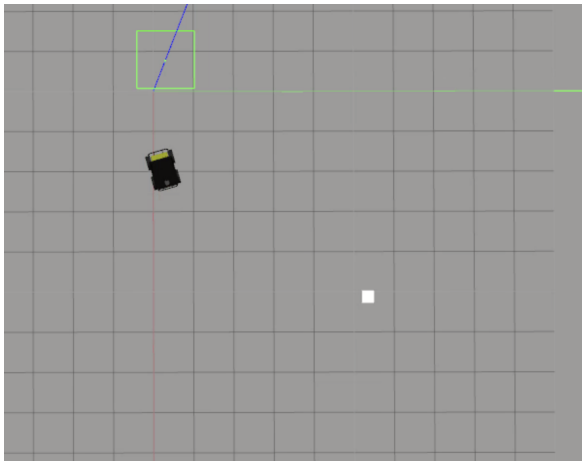


Figure 3.3: Empty world for random target reaching.

Table 3.3: State and action definition for target reaching.

Signal	**Elements**	**Dimension**
State	related position, distance, action of last time step, index	$R^{1\times 6}$
Action	linear velocity, angular velocity	$R^{1\times 2}$

In state signal, the related position refers to the difference between the current position of UGV and the target position without considering the Z axis, which is defined as

$$p_r(x,y) = p_{UGV}(x,y) - p_{target}(x,y), \tag{3.1}$$

where the value of $p_{UGV}(x,y)$ and $p_{target}(x,y)$ can be obtained from the Gazebo simulation. The related position not only has concise and informative representation about the position relationship between the robot and target, but also reduces the dimension of state and hence decreases the computational complexity. The distance is measured by calculating the euclidean distance between target and robot,

$$D = ||p_{UGV}(x,y) - p_{target}(x,y)|| = \sqrt{(p_{UGV}(x) - p_{target}(x))^2 + (p_{UGV}(y) - p_{target}(y))^2}. \tag{3.2}$$

The action of last time step is added to the state signal to make the state more informative and allow the robot to move more smoothly. The index is actually an indicator that can only be 0 or 1. If index is equal to 1, it means that the robot is close to the target position within a small threshold value, while 0 means the agent is far from the

target. This important element is used in designing the reward function, which will be described in the following section.

3.2.3 Reaching and Obstacle Avoidance

Now the obstacle are added to the environment and the accessible space is restricted to a square with $12m$ in both height and weight. To let the robot perceive the exist of obstacle, the signal from laser sensor must be used to measure the safety distance. However, the amount of raw laser beams is somehow very large, which is unsuitable to be used directly. Thus, the processed laser signal is obtained by discretizing the raw signal. The cover range is reset to $[-\pi/2, \pi/2]$ because we do not care about the information behind the UGV. Finally, only 10 beams are extracted from the raw sensor with equal distance and concatenating to the representation of state signal, which is now $s \in R^{1\times 16}$. The distance range of the sensor is set to $[0.3, 10]$,where $0.3m$ indicates the minimum tolerance distance between the UGV and obstacle. If any of the measured distance is smaller than this tolerance, the state can be seen as terminal state.

The obstacle are set up in the environment, which is shown in Figure 3.4.

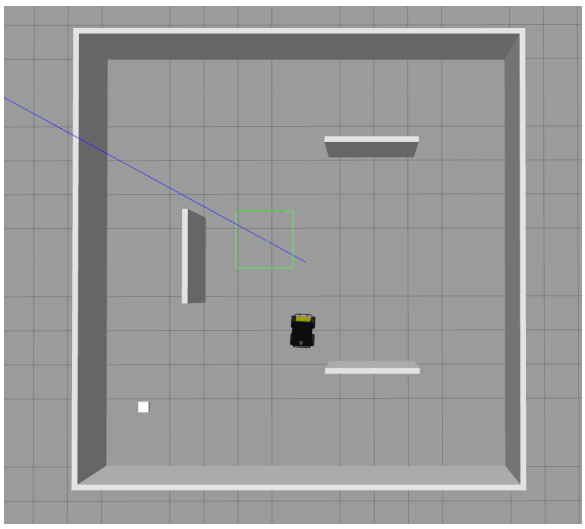


Figure 3.4: Environment with obstacle.

3.3 Algorithms

This section introduces the main algorithms that are used in the project, including optimization algorithms (SGD and Adam), and the DRL algorithms used in this project (DQN, DDPG and TD3).

3.3.1 Optimization algorithms

Deep reinforcement learning adopts the neural network from deep learning as function approximator, which is usually optimized using gradient descent based methods. Two typical and useful methods are used, one is Stochastic Gradient Descent (SGD) and another is called Adam.

SGD is a greatly improved method based on standard gradient descent. It calculates the gradient based on the loss computed by a random mini batch data. Thus, in each training episode, the back-propagation can be performed a lot of times, which dramatically increases the learning efficiency and accelerates the convergent speed. Algorithm 6 demonstrates the standard SGD algorithm for supervised learning. There are some

Algorithm 6 Stochastic Gradient Descent

Require: Learning rate η, label set $[y^1, ..., y^N]$
Require: Initial network parameter θ

while Stopping criterion not meet **do**
  Sample a mini-batch of m samples $[x^1, ..., x^m]$ from dataset
  Set $g = 0$ ▷ g is the gradient
  for t=1,m **do**
    Compute gradient: $g \leftarrow g + \nabla_\theta L(f(x^i; \theta), y^i)$
  $\theta \leftarrow \theta - \eta g$

differences when applying SGD in reinforcement learning. As mentioned in chapter 2, the label set must be provided for loss computation. Since in RL problems, there is no label set, we should find a replacement to represent the label, and this is why in most of DRL algorithms, the network always has its copy, which is called target network. The output of target network is used to replace the position of label and the convergent criterion is making these two network as close as possible. The data used for generat-

ing the predictions and their targets are transitions stored in the replay memory.

Adam is a more advanced optimization algorithm that proposed to tackle some limitation of SGD. The first problem is that the learning rate is fixed in SGD, which leads to the problem of divergence. When the model weights are very close to the optimal weights, it is unsuitable to use the same learning rate as initial. An intuitive solution is to reduce the learning manually when the model tends to convergence. Adam introduces another two adaptive variables that decay the gradient during training to avoid constant learning rate. Second, it uses two moment vectors to record the history of gradient so that the update direction can be more smooth and stable. Besides, this method also solves the problem of training the non-stationary data. In supervised learning, the training data is assumed to be stationary, which means that the distribution of the data is fixed. While in reinforcement learning, the policy is changed over the time, resulting in the change in distribution of transitions. Therefore, the moment vectors can keep track of historical transitions and stabilize the training performance.

Algorithm 7 Adam

Require: Learning rate α, denominator bias $\varepsilon = 10^{-8}$
Require: $\beta_1 = 0.9$, $\beta_2 0.999$; Exponential decay rates for moment estimation
Require: Initial objective function $f(\theta)$ with random weights θ_0

$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (initialize time step)
while Stopping criterion not meet **do**
    $t \leftarrow t+1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ Compute gradient at time step t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ (Update first moment vector)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_v) \cdot g_t^2$ (Update second moment vector)
    $\hat{m}_t \leftarrow m_t / (1-\beta_1^t)$
    $\hat{v}_t \leftarrow v_t / (1-\beta_2^t)$
    $\theta_t \leftarrow \theta_{t-1} - \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ (update parameters)

3.3.2 Implementation of DQN

The dueling DQN is used for obstacle avoidance. The architecture of Q-network is in Figure 3.5. 'dim' means the dimension of input state. Linear layer is the same with

the operation in ANN, which is linear transformation with the equation $y = wx + b$. The two parameters represent the input dimension and output dimension. Before the final computation of Q value, the similar mechanism of dueling network [4] is used. The separated two layers calculated the action value and state value, respectively, then the Q value of state-action pair is obtained by adding the state value to each action value. The action selection is based the ε-greedy policy. The learning process has been mentioned in Algorithm 4 and the hyper-parameters are listed in Table 3.4.

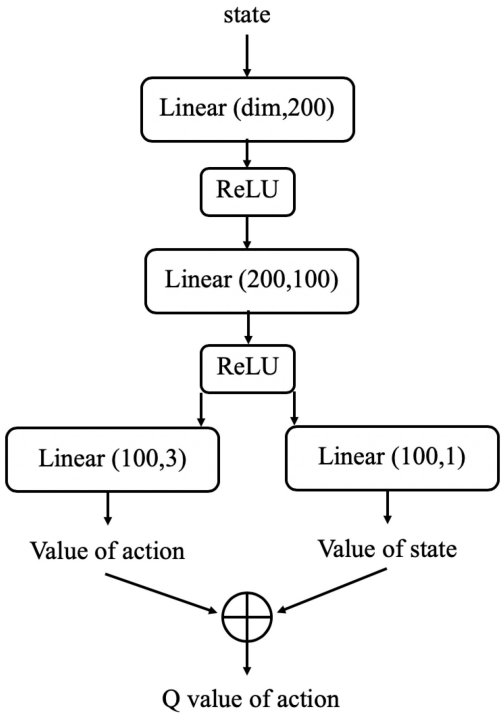


Figure 3.5: Q-network of Dueling DQN.

Table 3.4: Hyper-parameters of Dueling DQN.

Parameters	**value**
Optimizer	SGD
Learning rate	0.01
Batch size	32
Discount factor (γ)	0.9
ε	0.9
Target network update steps	10^3
Memory size	10^6

3.3.3 Implementation of DDPG

The Deep Deterministic Policy Gradient (DDPG) is used as the baseline algorithm in this research project. The network structures of actor and critic are illustrated in Figure 3.6.

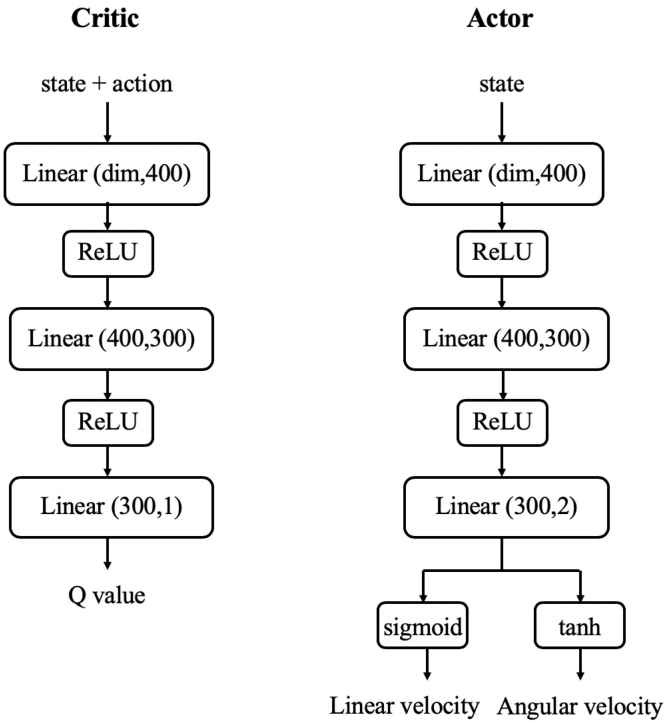


Figure 3.6: Network architectures of actor and critic.

The 'Linear' layer operates the linear transform to the input data (the first parameter is the input dimension while the second represents the output dimension). Between each linear layer is the activation function. The ReLU function is used for most output of linear layer except the output of actor network. For critic network, the input is comprised by both state and action signal. The action used for forming the state is last step action, while the action concatenates with the state as input to the critic is current action. The Q value is generated from the last linear layer without activation. For actor network, it is similar to the critic, but the output is separated into two velocities and activated by sigmoid and tanh function, respectively. Sigmoid function maps the input into a range of $[0, 1]$, while output of tanh function is lied in $[-1, 1]$. The plots of the function can be seen in Figure 3.7. The sigmoid function is used to activate the linear velocity, which is always expected to larger than 0. This operation forces the actor

output positive linear velocity so that the robot will never turn back. The tanh function generates the angular velocity such that the agent can turn right or turn left.

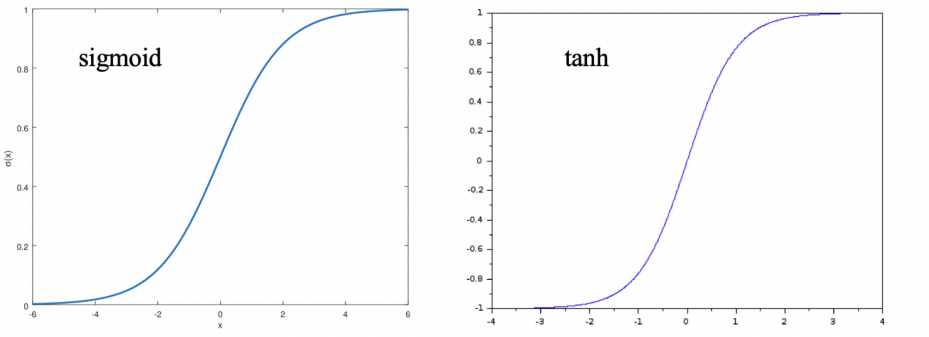


Figure 3.7: Plots of sigmoid and tanh function.

According to Algorithm 5, a random process is added to the action generated by the actor. The Ornstein-Uhlenbeck process, same with paper [28], is used as noise and added to the action for exploration. This random process is defined as below, where $\mu = 0$, $\theta = 0.15$ and $\sigma = 0.2$ The detail specification of networks and training process will be described in the following section.

Algorithm 8 Ornstein-Uhlenbeck process

Require: μ, θ, σ
Output: N

Initialize $N = [1,1] * \mu$
for episode=1,M **do**
  Reset N
  for t=1,T **do**
    $\delta \leftarrow \theta(\mu - N) + \sigma N(0,1)$
    $N \leftarrow N + \delta$

3.3.4 Implementation of TD3

While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyper-parameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function.

Twin Delayed DDPG (TD3) [30] is an algorithm which addresses this issue by introducing three critical tricks, including clipped double Q-learning, delayed policy updates and target policy smoothing.

TD3 learns two Q-functions by mean square Bellman error minimization, in almost the same way that DDPG learns its single Q-function. The target Q-value is computed using the smaller value computed from these two Q-functions. Before obtaining the target Q-value, a clipped noise is added to the target action, which is used for predicting the target Q-value. This operation avoids overestimation of standard Q-network and stabilizes the convergence of training. Different from DDPG, TD3 use the normal distribution process as noise, which is clipped with a range to avoid sharp change in action. Besides, the policy and the target networks, are updated every d steps rather than

Algorithm 9 TD3

Initialize replay memory D
Initialize critic network $Q(s,a|\theta^{Q_1})$, $Q(s,a|\theta^{Q_2})$ and actor network $\mu(s|\theta^{\mu})$ with random weights
Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q_1^{'}} \leftarrow \theta^{Q_1}$, $\theta^{Q_2^{'}} \leftarrow \theta^{Q_2}$ $\theta^{\mu^{'}} \leftarrow \theta^{\mu}$
for episode=1,M **do**
  Initialize a random process N for action exploration
  Initialize state s_1
  for $t = 1, T$ **do**
    Select action $a = \mu(s|\theta^{\mu}) + N(0,\sigma)$
    Execute action a and observe reward r and next state $s^{'}$
    Store transition $(s,a,r,s^{'})$ in D
    Randomly sample mini-batch of transition $(s,a,r,s^{'})$ from D
    $\varepsilon \leftarrow clip(N(0,\tilde{\sigma}),-c,c)$, $\tilde{a} = \mu(s^{'}|\theta^{\mu^{'}}) + \varepsilon$
    Set $y = r + \gamma \min_{i=1,2} Q^{'}(s^{'},\tilde{a}|\theta^{Q_i^{'}})$
    Update critic by minimizing the loss: $L = \arg\min_{\theta^{Q_i}} \frac{1}{N}\sum_i (y - Q(s,a|\theta^{Q_i}))^2$
    if t mod d **then**
      Update the actor policy using the policy gradient:
      $\nabla_{\theta^{\mu}} J \approx -\frac{1}{N} Q(s,\mu(s|\theta^{\mu})|\theta^{Q_1})$
      update the target network:
      $\theta^{Q_i^{'}} \leftarrow \tau\theta^{Q_i} + (1-\tau)\theta^{Q_i^{'}}$
      $\theta^{\mu^{'}} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu^{'}}$

each step, leading to a reduction in computation and improvement in convergence. The detailed algorithm can be seen in Algorithm 9. The architecture of actor and critic net-

works is the same with DDPG. The comparison of hyper-parameters selection between DDPG and TD3 is listed in Table 3.5.

Table 3.5: Comparison of hyper-parameters selection between DDPG and TD3.

Hyper-parameters	**DDPG**	**TD3**
Critic Learning Rate	10^{-3}	10^{-3}
Actor Learning Rate	10^{-4}	10^{-3}
Optimizer	Adam	Adam
Target Update Rate (τ)	10^{-3}	$5 \cdot 10^{-3}$
Batch Size	64	100
Discount Factor (γ)	0.99	0.99
Policy Update Rate (steps)	1	2
Target network Update Rate (steps)	1	2
Exploration Policy	$N(0, 0.1)$	OU process, $\mu = 0$, $\theta = 0.15$, $\sigma = 0.2$
Target Policy Noise	None	$N(0, 0.2)$
Noise clip (c)	None	$[-0.5, 0.5]$
Memory size	10^6	10^6

3.4 Reward Function

The definition of reward function is crucial in RL problem as it implicitly shows the goal of the problem. In some cases which have small and discrete state and action spaces, the reward function is set to sparse for fast convergence. This setting works well in simple environment and task, but is not suitable for complex environment, especially for optimal control problem, where the state and action space become infinite. The reason is that, for sparse reward function, the agent only receives a positive reward when it takes a significant action, and always receives zero when it explores the environment. If the searching space is finite and relatively small, it is easy for the agent to find the way to approach the goal. For the first task, obstacle avoidance, since the state and action space are both discrete, we define a sparse and simple reward function 10 for fast convergence. The reward is set to a negative value with two cases. The first is that the action is straight when the minimum distance is less than 1, which means that the action selection is bad. Another case is that when the robot crashes with the obstacle, the reward is assigned to a very high penalty.

Algorithm 10 Reward function: obstacle avoidance

Input:action a, state s and terminal condition t
Output: Instant reward r

if a is straight **then**
  if $\min(s) < 1$ **then**
    $r = -1$
  else
    $r = 1$
if a is turn left or turn right **then**
  $r = 0$
if t is Ture **then**
  $r = -100$

In control problem, it is difficult for agent to search the best action sequence that can approach the goal, leading to the problem of convergent speed. Therefore, the reward function should be dense and continuous for control problem. The basis component of the dense reward function is defined as

$$r = \alpha D + \beta ||a||, \tag{3.3}$$

where D and a are distance and executed action, respectively. α and β are two negative constants. The first part of the reward function is penalty of distance, while the second part is the penalty of action. This setting makes the reward negative in most of time. The closer the robot to the target, the larger the reward will be received. Besides, the penalty of action stabilizes the action and forces the robot to stop. By intuition, the robot should stop when it reach the target position. The penalty of action maximize the reward when the action is close to zero. The novel definition of reward function used for target reaching task is shown in Algorithm 11.

The terminal condition variable t aims to show that whether the agent finishes the task or reaches the terminal state. The counter C starts to increase once the distance is smaller than the tolerance, in which the reward is increased by 1. Furthermore, If the counter is larger than the predefined threshold value, which means that the agent has been stay within the difference tolerance for T steps, the reward is increased by T and

Algorithm 11 Reward function: Target reaching

Input: Distance D, action a, counter C and Threshold T (the required steps of stop when robot reach the target)
Output: Instant reward r, terminal condition t

$r = \alpha D + \beta ||a||$
$t = False$
if $D < \varepsilon$ **then** ▷ ε is the small tolerance
    $r \leftarrow r + 1$
    $C \leftarrow C + 1$
    if $C > T$ **then**
        $r \leftarrow r + T$
        $t = True$
        $C = 0$
else
    $C = 0$
if $D > d$ **then** ▷ d is the maximum tolerance distance
    $r = -1$
    $t = True$

reset the counter to 0. During the steps that the agent has reached the target position, if it does not stops and goes outside of the tolerance region, the counter is also set to 0. Finally, if the distance is too large, which means the robot goes far away from the target, we give -1 to reward and terminate the process. This mechanism allows the agent to stop when it has reached the target and dramatically accelerates the speed of convergence by restricting the accessible space of the robot.

As for the reward function used for obstacle avoidance, the main process is similar, while the difference is just the condition for termination. The laser signal d is used to determine whether the agent should be terminated. Besides, the reward is set to -10 when terminal to emphasize the importance of obstacle avoidance. The detailed definition is in Algorithm 12.

In next chapter, the detailed experiments specification will be described. The performance of these three tasks using the corresponding deep reinforcement learning algorithm and reward function will be illustrated. For different tasks, the specification is set differently to meet the best performance. The detailed analysis and discussion

Algorithm 12 Reward function: Target reaching and obstacle avoidance

Input: Distance D, laser signal $d \in R^{1\times 10}$, action a, counter C and Threshold T (the required steps of stop when robot reach the target)
Output: Instant reward r, terminal condition t

$r = \alpha D + \beta ||a||$
$t = False$
if $D < \varepsilon$ **then** ▷ ε is the small tolerance
    $r \leftarrow r + 1$
    $C \leftarrow C + 1$
    if $C > T$ **then**
        $r \leftarrow r + T$
        $t = True$
        $C = 0$
else
    $C = 0$
if $\min(d) < 0.3$ **then** ▷ $0.3m$ is the maximum tolerance distance
    $r = -10$
    $t = True$

related to some importance factors in RL problem will be included in chapter 5.

Chapter 4

Test and Experiments

In this chapter, the training specification and the performance of the reinforcement learning algorithm are included. The specifications and results are task-specific so that they are shown case by case.

4.1 Training specification

The training of robot using DRL algorithms is implemented using Python scripts. There are three types of scripts, including the agent script, the environment script and the main script. The agent script defines the main components of DRL algorithm, such as network structure, storing transitions, action selection and network optimization. Environment script sets up the model of environment, which includes the definition of state and action space, the execution of action, the initialization of the environment and the reward function. The main script integrates the agent and environment, and defines the whole training process for the reinforcement learning system. Although we use alternative algorithms for three different tasks, the training processes are similar and shown Algorithm 13.

Apart from keeping training the model, the evaluation is performed every 10 episodes. The evaluation process tests the current policy with also 10 episodes and returns the average cumulative reward. There are two differences compared with training. The first is that, there is no optimization in evaluation mode, and therefore, the model weights

are fixed. Another is that, the noise is removed from action selection. For DQN, the action is selected based on the maximum Q value of the state-action pair, while for DDPG and TD3, the action is generated purely by its deterministic policy.

Algorithm 13 Pseudo code: RL training procedure

Require: Initialize RL *model* and environment *env*

for $episode = 1, M$, **do**
  $s = env.reset()$ ▷ Initialize the environment
  $R \leftarrow 0$
  for $step = 1, T$ **do**
    $a = model.ActionSelcet(s)$ ▷ decision making
    $s^{'}, r, t = env.step(a)$ ▷ take action and receive next state and reward signal
    $model.StoreTransition(s, a, r, s^{'})$
    $s \leftarrow s^{'}$
    $R \leftarrow R + r$
    if $model.MemoryCounter > T$ **then**
      $model.learn()$ ▷ update model
      if $t == True$ **then**
        $s = env.reset()$

Basically, this training procedure is used for all three tasks in this project. In next several sections, the performance of the RL algorithm for specific task will be explained.

4.2 Obstacle Avoidance

The DQN algorithm 4 and reward function 10 are used for this task to generate discrete actions, including going straight, turning left and right. The objectives is just to keep moving while avoid any collision. The robot is always initialized on the top left of the maze in Figure 3.2. The number of steps in each episode is 200.

We train the DQN for 100 episodes and evaluate the progressive model for 10 times. The results are illustrated in Figure 4.1. The model tends to be convergent at about 70^{th} episode.

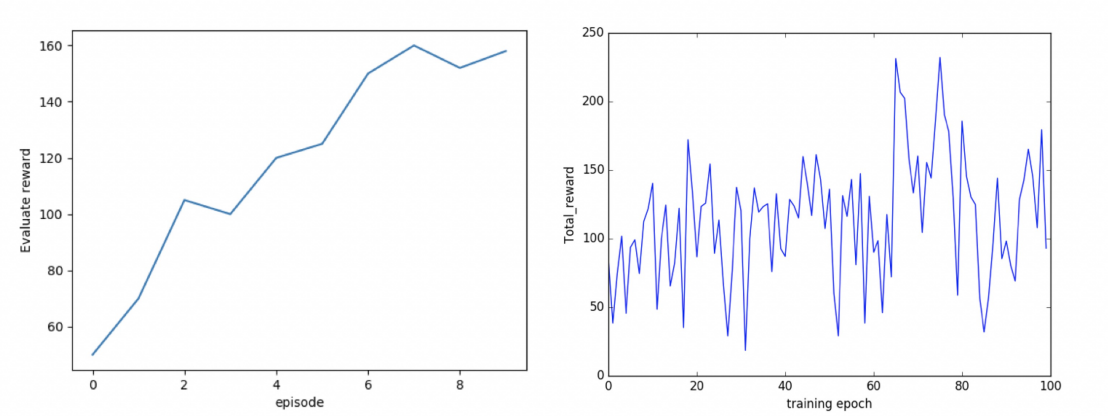


Figure 4.1: Training and evaluating cumulative reward for obstacle avoidance.

We attempt 3 trials of training with 100 episodes and test the performance of the model for 10 episodes after training in each trial. The quantitative performance of the model is represented by the mean cumulative reward and its variance. The result is shown in the Table 4.1 at the end of this chapter.

4.3 Random Target Position Reaching

The DDPG algorithm 5 and reward function 11 are used for target reaching to generates the continuous actions. Since the range of output of actor is within $[-1,1]$, but it is too large for the robot to control its motion. Thus, an action bound $[-0.5,0.5]$ is multiplied by the output of actor element-wise to limit the minimum and maximum action value, so that the robot can move stable inside the environment. The initial position is at the origin of the environment, where the robot is reset when it moves away from a square region with $10m$ in both height and weight. The target position is generated randomly after the position initialization within a range of $x,y \in [-5,5]$.

The number of training episodes is 150 for this task. The plots of reward is shown in Figure 4.2. Apart from visualizing the trend of reward, the change of gradient of actor and average Q value of critic are shown as in Figure 4.3. These two value are opposite based on equation in algorithm 9.

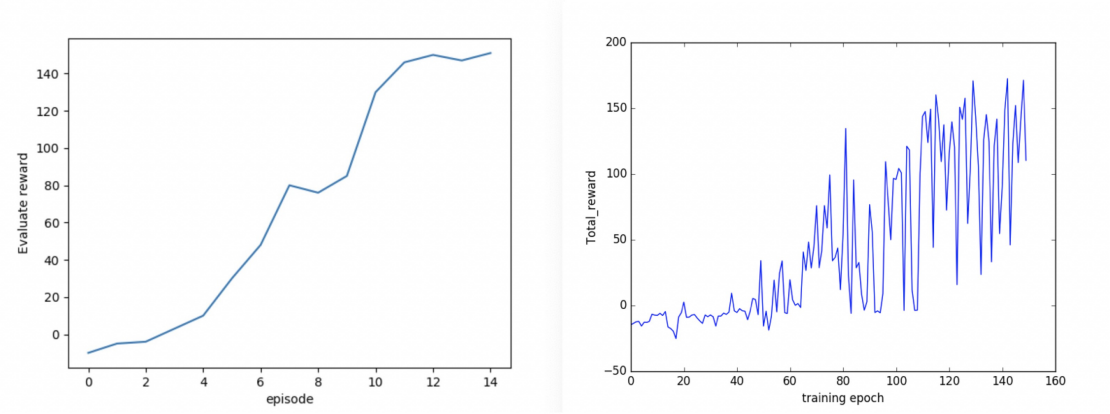


Figure 4.2: Training and evaluating cumulative reward from target reaching.

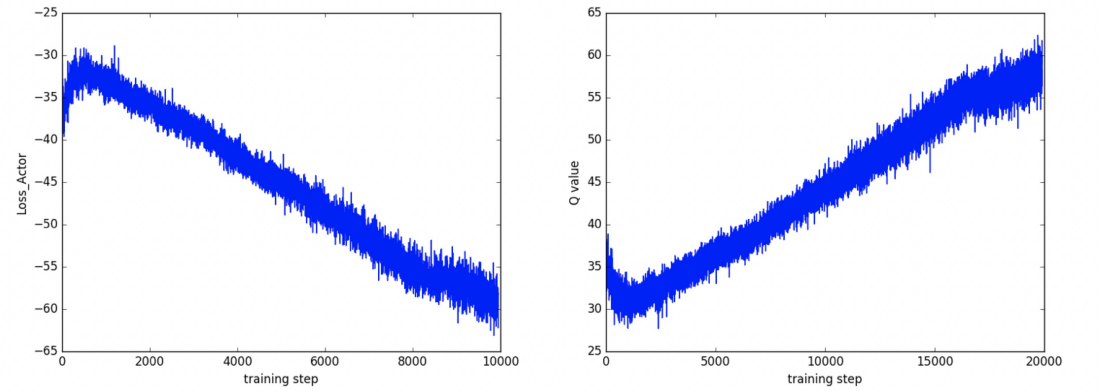


Figure 4.3: Gradient of actor and average Q value of critic. Left plot is loss and right plot is Q value.

We attempt 3 trials of training with 150 episodes and test the performance of the model for 10 episodes after training in each trial. The quantitative performance of the model is the same as before and the result is shown in the Table 4.1 at the end of this chapter.

4.4 Reaching and obstacle Avoidance

Both DDPG 5 and TD3 9 are used to train the robot for both target reaching and obstacle avoidance using the reward function 12. The reason why using TD3 for this task is that, when the robot is trained in the environment shown in Figure 3.4, the agent tends to converge for a while, then it diverges with the increased number of episode. This phenomenon is called overestimation of Q function, which is critic in DDPG. The

detailed discussion on why TD3 can relief this problem will be included in chapter 5. Since this task is more difficult than previous two tasks, the training episode is set to 500 for DDPG and 400 for TD3, respectively. The initial position is also the origin, and the target position is generated randomly within a limited region shown in Figure 4.4. The reward, loss of actor of DDPG and TD3 are illustrated in Figure 4.5.

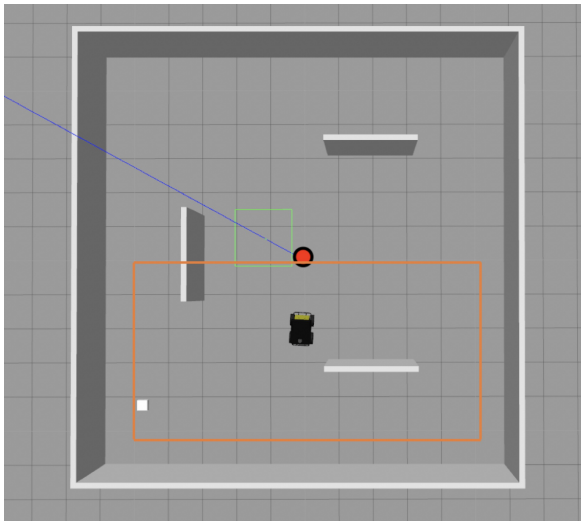


Figure 4.4: Environment of reaching and obstacle avoidance (region 1). The red point is the origin point, and the orange boundary is the region of random target position.

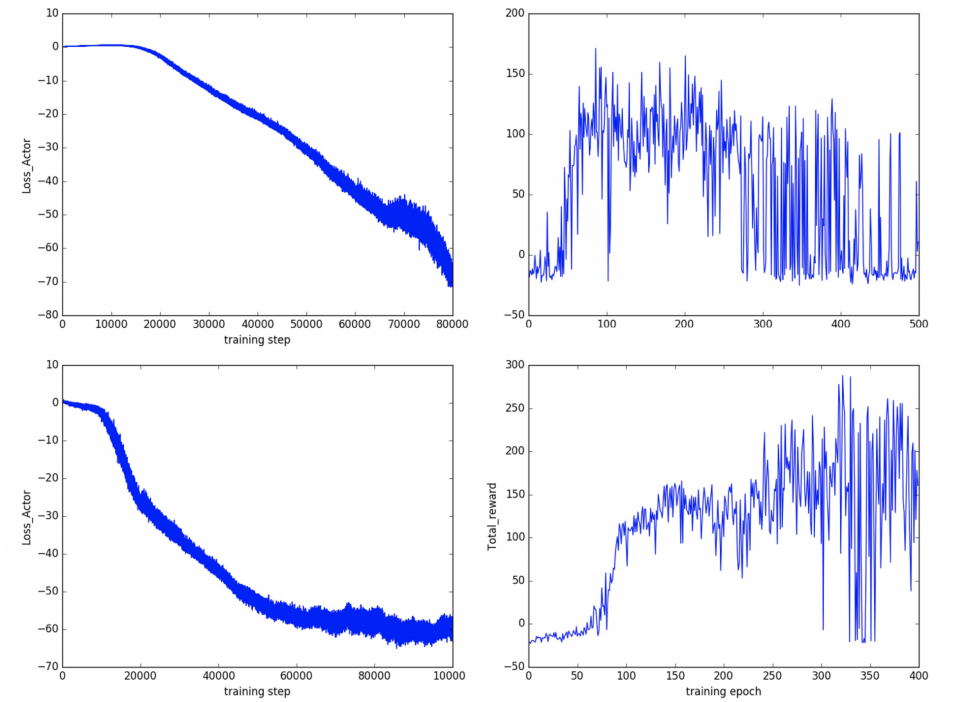


Figure 4.5: Gradient of actor and training reward. Top two plots are from DDPG, bottom are from TD3.

We repeat the evaluation process as previous tasks for both training of two random regions. The number of training episode is set to 250 to avoid overestimation of Q function. The overall performance of the models used for each task is shown in the Table below. It can be seen that, the performance of obstacle avoidance is relatively stable than other task. Performance of target reaching is definitely more superior than the reaching & avoidance and also stable over the three different trials. TD3 is absolutely better than DDPG in both optimal performance and stability.

Table 4.1: Overall performance of different trials for three tasks after training.

Task	**Algorithm**	**Trial 1**	**Trial 2**	**Trial 3**
obstacle avoidance	DQN	153.2 ± 10.7	149.8 ± 8.2	160.0 ± 8.7
Random target reaching	DDPG	143.0 ± 11.2	150.8 ± 7.9	144.5 ± 13.7
Reaching & avoidance	DDPG	101.2 ± 28.6	94.1 ± 30.3	110.5 ± 29.1
Reaching & avoidance	TD3	128.2 ± 18.1	137.8 ± 19.6	132.0 ± 16.3

Chapter 5

Discussion

In this chapter, the critical factors that affect the performance of training the RL system are discussed. The key factors are as follow,

- **Markov property of environment model:** affects the success of convergence directly.
- **Exploitation and exploration:** affects the speed of convergence.
- **Reward function:** affects both the success and speed of convergence.
- **Optimization method:** affects the success of convergence.
- **Overestimation of value function:** affects the success of convergence.

5.1 Markov Property of Environment Model

As mentioned in chapter 2, the environment has the Markov property if and only if its reaction to the agent only depends on current state and action, or in other word, the next state s_{t+1}, is only relied to the current state s_t and action a_t. The agent can make decision on action selection by giving only the current state. To satisfy this property, the state and action space should be defined carefully to represent the characteristic of the environment as full as possible

The action space is simple in this project. Two different action spaces are used to

meet the requirement of specific DRL algorithm. For DQN, the discrete action space is defined, which includes three elements: go straight, turn left and trun right. The linear and angular velocities are fixed for all three actions. The DQN is used for obstacle avoidance, which is explicit and relatively simple, and hence the three discrete actions can fully satisfy the Markov property in this task. For another two tasks, the robot need to arrive the target position and stop, and hence the smooth control is required. DDPG and TD3 generate the continuous action leading to a flexible control, which is more suitable for target position reaching.

The definition of state space is more ingenious and critical than action space. For obstacle avoidance, the state is represented by 5 laser beams, which measure the distance between the obstacle and robot. However, this definition is not suitable for task two and task three, because it does not contains any useful information about the position relationship. For target reaching, since the environment is empty, the laser signal is useless in this application and can be removed. According to Table 3.3 the six elements state is well defined to meet the Markov property. The first two are relative position between target and robot, which is a concise expression (based on equation 3.1) to indicate the position relationship over the dynamic motion of robot. The distance D is the key measure to evaluate the instant reward. The action a_{t-1} is just as a record to the last time step. The final element is the indicator that is set to 1 if the robot has reached the target position and 0 if not. This element makes the state signal more Markov and hence the increases the representing capacity of the state given the empty environment. The final task uses the combination state of previous two tasks so that it can achieve both target reaching and obstacle avoidance.

The definition of state is not unique and is reasonable as long as it satisfies the Markov property. Apart from the state definition used in this project, some different elements can be added to the state ,such as the difference of orientation, the progressive distance difference and etc.

5.2 Exploitation and Exploration

The trade-off between exploitation and exploration is a subarea in reinforcement learning research. Exploitation allows the agent to select action based on what it has learned. Exploration requires the agent to select the action randomly to find the possibility of the environment. To achieve the objectives, the agent must exploit the environment under the current policy and value function. While to learn these policy and value function, the agent must explore the environment so that it can find the good state-action pair scored by the reward signal. This is the reason why both of them should be included in the training of RL system.

DQN uses ε-greedy policy, for balancing exploitation and exploration. DDPG and TD3 use Ornstein-Uhlenbeck process and normal process, respectively. However, in this project, for each method, the proportion of exploration is unchanged during the time, because ε is a constant and the mean and variance of random process are also fixed. Therefore, the probability of exploitation and exploration of the agent is also not changed when the model tends to converge. This setting is not advisable enough. By intuition, when the agent starts to learn in the environment, it should spend more time in exploring rather then exploiting, so that it can collect varied transitions and learn from them. When the agent has learned a good policy (tend to converge), it should reduce its focus on exploration, and collect more good transitions under the current policy. A simple improvement is to decay the ratio of exploration of the agent during the training episode. For example, for DQN, the ε can be increased over the time. For DDPG and TD3, the mean and variance can also be reduced when the number of episode is increased. A more advanced method would be making the proportion of exploration decrease adaptively based on the learning situation of the model.

5.3 Reward Function

Reward function tells the agent what is good for achieving the goal in the environment. Therefore, it implicit describes the objectives of the agent and highly rely on the spe-

cific task. Reward function can be sparse or dense, depending on the problem we want to solve. For the first task, obstacle avoidance, the sparse reward function is used because the goal is intuitive and simple. According to algorithm 10, the reward is equal to 1 only when the action is going straight with minimum laser distance larger than 1. The -1 is given when the agent is crashed with the wall. This reward will encourage the agent to go straight in all of the time, while turn left or right when the obstacle are very close.

The dense reward function is used for target position reaching and reaching while obstacle avoidance, which are very concrete and informative. Since now the action space is infinite, the action can be any value within the range. Thus, the sparse reward is hard to be applied to the continuous action and not suitable to be used in such cases. The basic reward function equation 3.3 used for task 2 and task 3 with α and β equal to -0.01. This setting allows the smooth change in received reward and enable the agent to learn a smooth control policy. The norm of action in the reward equation is utilized as the regularization term to avoid sharp change in action and sensitive to the position that can stop the agent when it is close to the target position. Other than using the reward equation 3.3, a trick is used to further increase the capacity of the reward function, which is shown in algorithm 11, 12. This mechanism combines the sparse reward to the dense reward and hence accelerate the convergent speed.

5.4 Optimization Method

Optimization is perhaps the most crucial part in training a DRL system. The most typical and prevalent method used in deep learning is gradient descent, in which the weights are updated by computing the back-propagation gradients from the loss. The loss is obtained by MSE between the prediction and the target. This is why in DRL algorithm, the value function has its target with the same structure and initialization. Different from the evaluate network that uses gradient to update its weights, the target network is updated by copying the weights from evaluate network directly. When the value function approximates the optimal value function, the loss is minimum and the

weights of these two networks are very close. The process of optimization can be seen as the searching of global minimum of the loss function.

In our project, the SGD 6 is used in DQN for obstacle avoidance and Adam 7 is used in DDPG and TD3. Adam is advanced than SGD in two domains. The first is that, Adam uses two decay parameters to reduce the magnitude of the gradient so that the loss function can approach its optimal minimum without divergence. Another is that, Adam uses two moment vectors to record the historical gradient direction so that the direction of the gradient will not be changed sharply, leading to a more stable training. However, these two mechanisms increase the probability of getting stuck in local minimum of loss function, such that the network cannot approximate the optimal network. Although SGD has drawbacks as previous mention, it is good at jumping out of the local minimum of loss function and find the direction of global minimum. Besides, the convergent speed of SGD is usually faster than Adam, for it uses fixed value of learning rate.

Another difference is that, SGD is used for training data with stationary distribution. In supervised learning, it is assumed that the volume of training data is unchanged in most of cases. But sometimes, if we want to add more data to the dataset and use the pre-trained model, the distribution is varied due to the adding of new data. Adam is suitable for training the dataset with non-stationary distribution by introducing the moment vectors to record the history of the gradient. In our project, the DQN is trained using SGD for obstacle avoidance. The task is simple and the action space is limit, so we assume that the transition distribution of the replay memory is stationary. Adam is used for the other two tasks.

5.5 Overestimation of Value Function

Overestimation is a severe problem in DRL problem. We found this problem when the RL system is trained continuously and then the model tends to work bad and divergence. The overestimation is mainly caused by the large Q value of the critic network

which is still increased when the policy (or actor network) is convergence. Since the update of actor uses the Q value of critic as gradient, which is increased without reaching a steady state, the weights of actor is always changed by performing gradient descent optimization and hence cannot approximate the optimal policy. Figure 4.5 shows the comparison of DDPG and TD3 used in task 3. The model that is trained using DDPG diverges when the training epoch is larger than 250 episode, and the magnitude of the gradient of actor (which also can be seen as the average Q value of critic) is continuously increased. TD3 can relief this problem to some extent by using three mechanisms, which has been mentioned in chapter 3.

However, TD3 cannot completely solve the overestimation of value function, because it just try to use some techniques to delay the update speed or increment of Q value of critic. By looking at the bottom two plots in Figure 4.5, the cumulative reward tends to oscillate with high variance when the training episode is large, resulting in the difficulty of convergence. Therefore, finding a robust method that can eliminate the overestimation is a promising topic in deep reinforcement learning domain.

Chapter 6

Conclusion and Recommendations

6.1 Conclusions

The objectives of this research project, divided into three steps, including obstacle avoidance, random target reaching and combination of first two, has been successfully implemented using deep reinforcement learning. The DQN is used for obstacle avoidance, DDPG is used for random reaching and both DDPG and TD3 are used to achieve the final objectives. Three different reward functions are designed to suit the characteristic of the corresponding environments. After training the reinforcement learning models for a number of episodes, the agent tends to convergent and learn the optimal policy. However, when training the third task, the problem of overestimation exists and makes the agent divergent with increased training episode. TD3 is used aiming to solve this problem, but it cannot completely eliminate the overestimation and just decay the degree of this phenomenon and delay it to the future.

6.2 Recommendations

Some interest and challenging directions of using deep reinforcement learning to the continuous control problem can be researched in the future. The first is to improve the balance between exploitation and exploration strategy. We use an unchanged strategy for this trade-off in this project, which is acceptable but not good enough for fast convergent property. A more advanced method for adjusting the proportion of both

exploitation and exploration should be researched and experimented. Another is solve the problem of overestimation by modifying the value function itself or the calculation of actor gradient. Besides, recalling that the transitions of historical experience is randomly sampled from the replay memory, which is very inefficient, especially when the size of the memory is large. By intuition, the agent should learn from the good experience other than the bad experience. However, the random sample method treats them as equal, resulting in a slow convergence. Thus, mechanism about improving the sampling efficiency of historical transitions could be researched in the future.

The works done in this project are relatively simple and ideal. In the future, the more complicated environments with high frequency of dynamics can be used to train the UAV using DRL and explore more problems and limitation of the applications of DRL in continuous control domain.

Bibliography

[1] Scott Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1):6, 2017.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[5] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36. IEEE, 2017.

[6] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3357–3364. IEEE, 2017.

[7] Oleksii Zhelo, Jingwei Zhang, Lei Tai, Ming Liu, and Wolfram Burgard. Curiosity-driven exploration for mapless navigation with deep reinforcement learning. *arXiv preprint arXiv:1804.00456*, 2018.

[8] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter Corke. Towards vision-based deep reinforcement learning for robotic motion control. *arXiv preprint arXiv:1511.03791*, 2015.

[9] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

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

[11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[12] Mohammad Abdel Kareem Jaradat, Mohammad Al-Rousan, and Lara Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1):135–149, 2011.

[13] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[14] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[15] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[16] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning. *ICML (3)*, 28(1139-1147):5, 2013.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[22] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[26] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[27] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[28] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[30] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.