UNIVERSITY OF SOUTHERN DENMARK

MASTER THESIS

Methods and implementations for coordinated multi-agent learning.

Author:
Kun QIAN

Supervisor:
Robert BREHM

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

Mechatronics with Profile in Embedded Control Systems
Mads Clausen Institute

June 2018

Declaration of Authorship

I, Kun QIAN, declare that this thesis titled, 'Methods and implementations for coordinated multi-agent learning.' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

*"Learning without reasoning will gain nothing but confusion. Thinking without learning will gain nothing but mental fatigue (*学而不思则罔，思而不学则殆*)."*

Confucius

UNIVERSITY OF SOUTHERN DENMARK

Abstract

Mechatronics with Profile in Embedded Control Systems
Mads Clausen Institute

Master of Science

Methods and implementations for coordinated multi-agent learning.

by Kun QIAN

In a multi-agent system (MAS), the agents choose actions independently and solely according to local sensor information. This decentralized control manner requires that the agents should be able to communicate with each other so that cooperation, coordination, and negotiation can be achieved. Moreover, in a MAS, learning is needed since it is impossible to specify optimal actions for all situations that an agent might encounter. Thus, reinforcement learning (RL) is tied to the concept of agent. The goal of this thesis is to investigate and evaluate suitable technologies and architectures for implementation of an collaborative and coordinated multi-agent system. This involves a research on the state-of-the-art in collaborative and coordinated multi-agent learning algorithms as well as research and evaluation of implementations and architectures for agent-based systems on the bases of a laboratory benchmark model.

The thesis first explains the mathematical models for RL and basics for agent-based systems. An agent-based system is then implemented for demonstrating the benchmark model, inside which, the control of rolling robots is purely based on a RL method. The experimental result shows that these robot agents can coordinate with each other to fulfill the task that requested by another agent.

Acknowledgements

First, I sincerely thank Prof. Robert Brehm for enabling me to write my master's thesis on this interesting topic. Without his advice, inspiration, and dedicated involvement, this thesis would not have been accomplished.

I would like to thank Prof. Lars Duggen, with whom, I discussed a lot about the mathematical theory and MATLAB implementations.

I would like to thank fellow students at Mads Clausen Institute. I really appreciate the chance to know all of them. I would like to thank particularly Anders Kristensen, Asfand Yar Khan, Berk Dikici, Christian Mørk-Pedersen, Ilian Pappa, Jegvan Jon Hansen, and Shayan Hajghasem, for their valuable discussions and for their priceless company during the writing.

I would like to thank Pu Wang, Shaodan Li, Shiyu Peng, Xi Liu, Xuan Lyu, and Ying Gao. With their company, I feel more comfortable staying in a foreign country. Additionally, I would like to thank Shiyu Peng again for the proofreading of my thesis and all her valuable suggestions.

Most importantly, none of this could have been possible without the support of my family. I feel fortunate to be born in this family.

Contents

List of Figures

List of Tables

Abbreviations

RL	**R**einforcement **L**earing
MAS	**M**ulti-**A**gent **S**ystem
MARL	**M**ulti-**A**gent **R**einforcement **L**earing
JADE	**J**ava **A**gent **DE**velopment
ROI	**R**ange **O**f **I**nterest
FIPA	**F**oundation for **I**ntelligent **P**hysical **A**gent
API	**A**pplication **P**rogramming **I**nterface
MDP	**M**arkov **D**ecision **P**processe
DP	**D**ynamic **P**rogramming
TD	**T**emporal-**D**ifference
AC	**A**ctor-**C**ritic
MG	**M**atrix **G**ame
SG	**S**tochastic **G**ame
MADRL	**M**ulti-**A**gent **D**eep **R**einforcement **L**earing
MAAC	**M**ulti-**A**gent **A**ctor-**C**ritic
ACL	**A**gent **C**ommunication **L**anguage
AP	**A**gent **P**latform
AMS	**A**gent **M**anagement **S**ystem
DF	**D**irectory **F**acilitator
MTS	**M**essage **T**ransport **S**ervice
IDE	**I**ntegrated **D**evelopment **E**nvironment

Chapter 1

Introduction

Agent technology has been considered one of the most popular technologies for industrial applications, especially for distributed systems [6, 7]. An agent needs to find out what to do to reach its goal rather than constantly being told [8], and thus, reinforcement learning (RL) is tied to the concept of agent [5]. A RL agent learns by trial-and-error interaction with its dynamic environment; while the agent interacts by executing actions, the state of the environment changes from one to another, and the agent uses the concept of state and iteratively propagates the reward it receives at the end to the chosen actions so that it can learn how to make decisions while performing the task and interacting with the environment [1, 5, 8]. A single agent with RL can be applied to many domains. For example, in the game of Go, the AlphaGo Zero achieved superhuman performance with only RL algorithm, winning 100–0 against the champion-defeating AlphaGo [9].

Although in many cases, agents can act individually to achieve their goals, several different agents integrated into a complete system also need to cooperate, coordinate, and negotiate with one another to cope with a complex problem involving either distributed data, knowledge, or control. This brings the emergence to multi-agent system (MAS). A MAS consists of many agents interacting with each other in the same environment [8]. Integrating RL methods into MASs has also attracted increasing attention recent years [10, 11] since the complexity of some tasks make it hard to solve with preprogrammed agent behaviors. The applications of multi-agent reinforcement learning (MARL) ranges from game playing to industrial applications. [12, 13] introduce a multi-agent robot scenario, in which learning is required to specify optimal actions for all states that each robot might encounter. [14, 15] introduce a set of MARL systems for traffic lights control, which help optimize the driving policies. [16] presents the optimization of distributed energy management using MARL approach.

Apart from the learning capability of a MAS, the architectures of an agent-based system are also of great importance, and an important topic is the development of programming languages and tools for the implementations. Java agent development (JADE) platform is widely used for facilitating the development of MASs both in research activities and in industrial applications [17].

The goal of this thesis is to investigate and evaluate suitable technologies and architectures for implementation of collaborative and coordinated multi-agent learning algorithms in Industry 4.0 applications. This involves a research on the state-of-the-art in collaborative and coordinated multi-agent learning algorithms as well as research and evaluation of implementations and architectures for coordinated multi-agent learning on the bases of a laboratory benchmark model.

1.1 Problem Definition

One of the most popular problems that is considered in the MASs is the formation control, where the aim is to form a prescribed geometrical shape in a given work-space without bumping into obstacles or colliding with each other. For example, for a cooperative navigation [18] problem, agents situated in an environment need to operate through physical actions to reach a set of landmarks. The agents must cover all landmarks without bumping into each other so that the task is then finished. This thesis will focus on how to solve this cooperative navigation problem on the bases of a laboratory benchmark model.

Benchmark Model

The setup of the laboratory benchmark model is shown in Figure 1.1. There is a black

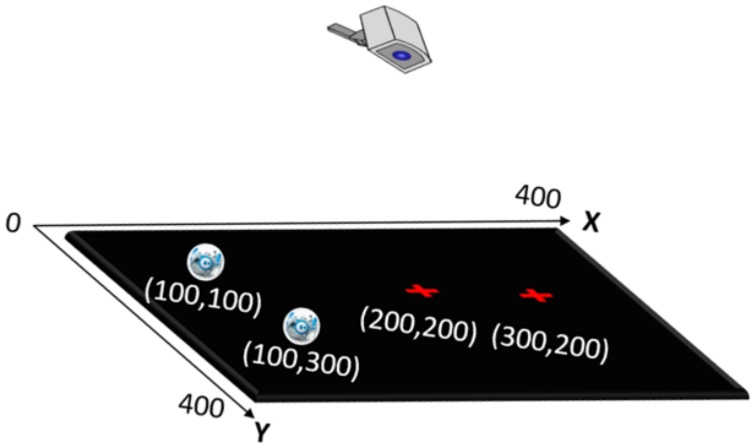


Figure 1.1: Setup of benchmark model.

area on the floor, above which, a camera is fixed and its range of interest (ROI) is set to be 400×400. There are two landmarks located at $(200, 200)$ and $(300, 200)$. The rolling robots, Spheros, situated originally at two stations located at $(100, 100)$ and $(100, 300)$ in this environment, know the relative positions of each other and also the landmarks. The robots need to cover the landmarks while avoiding each other. When the task is finished, they need to move back to stations to get charged. The chosen Sphero is SPRK+ (https://goo.gl/uvXsfs), which can be controlled through Bluetooth Low Energy. [19] has developed the API for controlling this Sphero.

1.2 Research Goal

The goal of the thesis is to investigate and evaluate suitable technologies and architectures for implementation of collaborative and coordinated MAS. Therefore, the following questions are addressed:

- What are appropriate RL algorithms for a MAS?
- What are the suitable architectures for developing MASs?
- How to integrate a RL method into a MAS?

1.3 Approach

The first part of the thesis focuses on implementing and evaluating RL algorithms for single agent and multi-agent environments. The second part will discuss the architectures for MASs and the integration of a RL method into a MAS.

Model-free RL algorithms have been applied to many domains, with which, the agent situated in the environment can make a decision without knowing the model of the environment. Traditional RL methods contain value table for each state so that the best action can be selected according to the values. One of the drawbacks of these methods is that it will not be practical when there are many states in the environment, not only because of the memory needed for large tables, but the time and data needed to fill them accurately [1]. To scale it to practical problems, function approximation methods based on the state features can be applied. These methods are especially needed for MASs, where the agent number also affects the size of the value table.

JADE is a software framework for developing agent-based systems which are compliant with the Foundation for Intelligent Physical Agents (FIPA) standards. The abstraction of software agent implemented in Java provides a simple application programming interface (API) [20]. The design of an agent in an agent-based system can be either coupled or embedded. The coupled design is popular in current automation scenarios, while the embedded design has many advantages like decoupling agents logically [21].

In this thesis, the investigation of different RL algorithms for single agent and multi-agent environments will first be done. Then the designs of software agents and agent-based systems will be studied. Based on the above knowledge, a cooperative navigation problem will be solved by integrating a RL method into the MAS.

1.4 Contribution

The main contributions of the thesis are:

- From single agent to multi-agent, the implementations in MATLAB for the chosen algorithms. Without using the existing toolbox, the step-by-step implementations help to understand the mathematical background.
- Applying RL to a MAS developed with JADE platform. This provides a strong insight of how to bring control methods based on reinforcement learning into agent-based systems.
- Controlling the rolling robots to fulfill tasks with parameters learned from simulated experience. This supports that theoretical research with simulations can be applied into real applications.

1.5 Thesis Outline

The remainder of the thesis is structured as follows:

Chapter 2 explains the formal mathematical model of RL. The chapter also discusses the implementations and evaluations of some popular RL approaches for single agent environment.

Chapter 3 provides a formal introduction to MARL. The chapter also introduces RL algorithms for a MAS. Implementations for some algorithms will be presented at the end.

Chapter 4 discusses the architecture for software agent, agent-based systems, and also the design for laboratory benchmark model.

Chapter 5 presents the implementation of laboratory benchmark model and the result of the experiment.

Chapter 6 concludes the work done in the thesis and provides directions for future work.

Chapter 2

Single Agent Reinforcement Learning

In this chapter, basics and solutions for single agent RL are discussed, which are fundamentals for MARL. The chapter begins by describing the elements and model for RL together with a self-defined simulation task in MATLAB. Then the implementations and evaluations for some popular methods described in [1] are presented.

All the implementations are done in MATLAB and can be accessed through the link: `https://goo.gl/8REkDz`. Finally, a short conclusion is drawn and suggestions for MARL methods selection are given.

2.1 Reinforcement Learning

RL is that an agent situated in an environment learns what to do or which action to take in a state so as to maximize the total reward it receives over the long run starting from this state. The agent discovers the best action by learning through trying them instead of being told.

Four basic elements can be identified while the agent is interacting with the environment: a policy, a reward signal, a value function, and, optionally, a model of the environment [1]. A policy gives the probability distribution over all possible actions in a state. Once the action is taken, the environment will send a numerical reward to the agent, which immediately indicates if the action chosen is good or bad. For example, for an agent to learn to find the landmark without hitting the obstacle in a certain area, the rewards can be $+1$ when reaches the landmark, -1 for hitting the obstacle, and 0 for all other positions. The value function specifies the expected accumulated reward over the future, which indicates the long-term desirability of a state. A model of the environment gives the dynamics of the environment that are useful, yet not necessary for finding the optimal policies, which leads to model-based and model-free learning. And in this thesis, only model-free learning methods are to be discussed.

Finite Markov Decision Processes

Finite Markov decision processes (MDPs) are a mathematically idealized form of RL problems. The agent perceives the environment, after decision being made, it takes an action, which leads to the state transition of the environment and it also gets a reward, as shown in Figure 2.1. In a finite MDP, there is a finite number of elements for a set of states ($\mathcal{S}^+$), actions ($\mathcal{A}$), and rewards ($\mathcal{R}$). And the states ($\mathcal{S}^+$) consists of non-terminal states ($\mathcal{S}$) and one or more terminal states. The agent interacts with the environment following discrete time steps, $t = 0, 1, 2, ..., T$. At a certain time step t, the agent perceives the environment's state, $S_t \in \mathcal{S}$, after decision being made, it takes an action, $A_t \in \mathcal{A}(s)$. Then at time step $t+1$, the agent receives a numerical reward, $R_{t+1} \in \mathcal{R}$, and perceives the environment's new state, S_{t+1}, until it reaches one terminal state at final time step T, which results in the sequence:

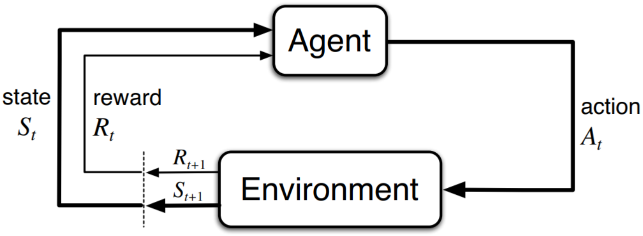


FIGURE 2.1: The agent-environment interaction in a Markov decision process [1].

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, ..., R_T, S_T. \tag{2.1}$$

This only makes sense in applications where there is a terminal state or states which will break the repeated interaction between agent and environment into episodes. The dynamics of the environment are defined by:

$$p(s^{'}, r \mid s, a) = Pr\{S_t = s^{'}, R_t = r \mid S_{t-1} = s, A_{t-1} = a\}, \tag{2.2}$$

for $s^{'} \in \mathcal{S}^+, s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$, which shows the probability of $s^{'}$ and r occurring at time step t given preceding state s and action a.

As mentioned above, the goal of an agent is to maximize the total reward it receives over the long run starting from a time step t, and the rewards after time step t is

$R_{t+1}, R_{t+2}, ..., R_T$. So, the simplest way is to maximize the sum of them:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T. \tag{2.3}$$

To encourage the agent to reach the goal faster, a discount factor γ can be introduced, where $0 \leq \gamma \leq 1$, for discounting the future rewards:

$$\begin{aligned} G_t &= \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-(t+1)} R_T \\ &= \sum_{k=0}^{T-(t+1)} \gamma^k R_{t+k+1} \end{aligned} \tag{2.4}$$

One property for the return value is:

$$\begin{aligned} G_t &= \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-(t+1)} R_T \\ &= \gamma^0 R_{t+1} + \gamma^1 G_{t+1}, \end{aligned} \tag{2.5}$$

which shows the recursive relations between returns at two successive time steps.

2.2 Single Agent Task Description

The environment for single agent experiments is defined in MATLAB as shown in Figure 2.2: inside an area of size 30 × 30, one agent is located at $(5, 5)$ while there are a goal located at $(25, 25)$ and an obstacle located at $(15, 15)$. The agent is moving so that it can reach the goal without colliding with the obstacle, and the states of the environment are denoted by the location of the agent. So, the non-terminal states are $\mathcal{S} = \{(1,1), (1,2), \cdots, (30,30)\}$, except two states $\{(15,15), (25,25)\}$, which are terminals states with reward -1 and $+1$, denoting the obstacle and goal respectively. There are four possible

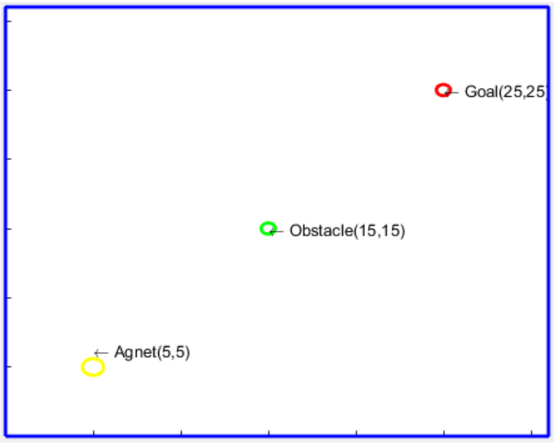


FIGURE 2.2: Single agent environment defined in MATLAB.

actions in each state, $\mathcal{A} = \{up, down, right, left\}$, and each will cause a non-deterministic result: the agent will end up at the correct heading with 80% probability and 10% for both relative right and left heading. If the heading is taking the agent out of the area, the state will remain unchanged. Thus, for instance, $p\left((25,25), 1 \mid (25,24), up\right) = 0.8$, $p\left((15,15), -1 \mid (15,14), right\right) = 0.1$, and $p\left((30,30), 0 \mid (30,30), right\right) = 0.9$. It is an episodic task and future reward is discounted by the discount factor γ. The reward is always 0 unless the agent reaches one of the terminal states. And the agent needs to find an optimal way to reach the goal without hitting the obstacle.

2.3 Solutions for Single Agent Environment

There are many ways for solving the above-defined task. Since the dynamics of the environment are given, the simplest way to solve it is to use dynamic programming (DP) methods, which compute the optimal policies for all states. However, a perfect model of an environment is not always available in practical applications, and the other methods to be presented can then be used to achieve the same effect.

As mentioned before, there are value functions for all states, which are used to estimate how much reward the agent can receive over the long run. Value functions are actually defined with respect to a policy, which gives the probability distribution over all possible actions in a state. For instance, the agent following policy π at time step t, $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$. And the value function of this state s, following this policy π, is defined by:

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-(t+1)} \gamma^k R_{t+k+1} \mid S_t = s\right], \tag{2.6}$$

for all $s \in \mathcal{S}$, where $\mathbb{E}_\pi[\cdot]$ denotes the expected value when the agent follows the policy π. Note that the values of terminal states should always be zero [1]. Similarly, the value of taking an action a in a state s and thereafter following policy π is defined by:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-(t+1)} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right], \tag{2.7}$$

which is called the *action-value function for policy* π. The relationship between these two functions is that the state value is the expectation of the action values for this state:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}\left[q_\pi(s,a)\right] \\ &= \sum_a \pi(a \mid s)\, q_\pi(s,a), \end{aligned} \tag{2.8}$$

for all $a \in \mathcal{A}(s)$. Similar to the property of the return value shown in (2.5), the value functions also satisfy recursive relationships:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p\big(s', r \mid s, a\big) \Big[r + \gamma \mathbb{E}_\pi\big[G_{t+1} \mid S_t = s'\big]\Big] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p\big(s', r \mid s, a\big) \Big[r + \gamma v_\pi\big(s'\big)\Big], \end{aligned} \tag{2.9}$$

$$q_\pi(s,a) = \sum_{s',r} p\big(s', r \mid s, a\big) \Big[r + \gamma v_\pi\big(s'\big)\Big], \tag{2.10}$$

which are used throughout RL, where good policies are to be searched. Following a good policy, or the optimal policy, denoted by π_*, optimal action values and state values will be achieved:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right], \tag{2.11}$$

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_*(s,a) \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma v_*\big(s'\big)\big], \end{aligned} \tag{2.12}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$.

2.3.1 Dynamic Programming

To organize and structure the search for an optimal policy for an agent situated in an environment, value functions can be used directly if the dynamics of the environment is known. For the above-defined environment, the dynamics $p\big(s', r \mid s, a\big)$ is given so that DP algorithms can be applied to improving approximations of the desired value functions, which give the optimal policies.

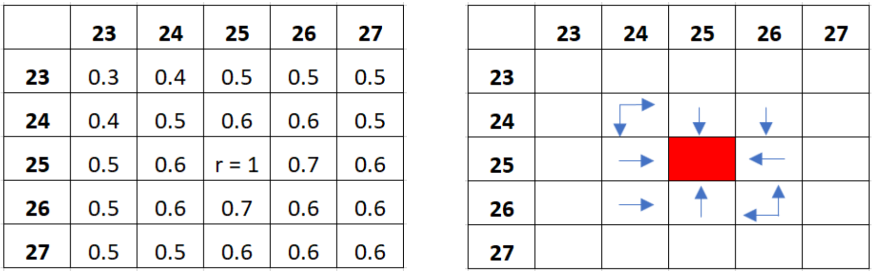
Policy Evaluation

For an arbitrary given policy π, the value of the current state is updated based on an expectation over all possible next states:

$$v_{k+1}\left(s\right) = \sum_{a} \pi\left(a \mid s\right) \sum_{s^{'},r} p\left(s^{'}, r \mid s, a\right) \left[r + \gamma v_k\left(s^{'}\right)\right], \tag{2.13}$$

for all $s \in \mathcal{S}$, which will give rise to a sequence of approximated value functions v_0, v_1, v_2, ... , v_k and it converges to v_π as $k \to \infty$.

Following the policy evaluation algorithm in [1], a random policy was created for all non-terminal states, i.e. 25% chances for each action, and the expected values are then calculated for all non-terminal states. Figure 2.3 and Figure 2.4 show the values and indicated policies around the goal and the obstacle, where the row and column indexes together indicate the location of the agent, i.e. the state of the environment. State $(25, 25)$ is the terminal state with the reward $r = 1$, indicated by the red block in Figure 2.3(b), and State $(15, 15)$ is also the terminal state with the reward $r = -1$, indicated by the green block in Figure 2.4(b). With the estimated values for the states, the optimal policies can then be calculated, indicated by the arrows in Figure 2.3(b) and 2.4(b). For example, for state $(24, 25)$, with the given dynamics $p\left(s^{'}, r \mid s, a\right)$, the values for actions $\{up, down, right, left\}$ are $\{0.51, 0.55, 0.75, 0.91\}$, which means the optimal action is *down*.



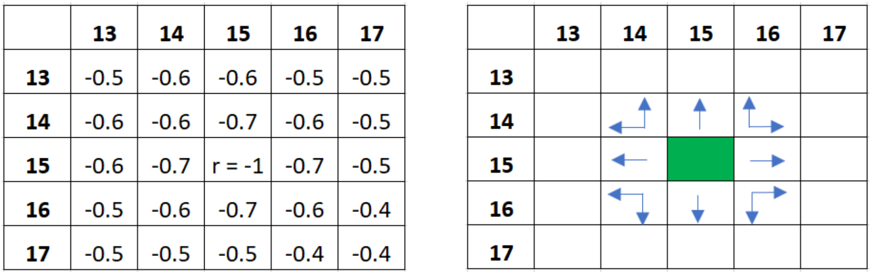
	23	24	25	26	27
23	0.3	0.4	0.5	0.5	0.5
24	0.4	0.5	0.6	0.6	0.5
25	0.5	0.6	r = 1	0.7	0.6
26	0.5	0.6	0.7	0.6	0.6
27	0.5	0.5	0.6	0.6	0.6

(a) Values.

	23	24	25	26	27
23					
24		→ ↓	↓	↓	
25		→	🟥	←	
26		→	↑	← ↑	
27					

(b) Indicated policies.

FIGURE 2.3: Values and indicated policies for states around the goal.



	13	14	15	16	17
13	-0.5	-0.6	-0.6	-0.5	-0.5
14	-0.6	-0.6	-0.7	-0.6	-0.5
15	-0.6	-0.7	r = -1	-0.7	-0.5
16	-0.5	-0.6	-0.7	-0.6	-0.4
17	-0.5	-0.5	-0.5	-0.4	-0.4

(a) Values.

	13	14	15	16	17
13					
14		← ↑	↑	↑ →	
15		←	🟩	→	
16		← ↓	↓	↓ →	
17					

(b) Indicated policies.

FIGURE 2.4: Values and indicated policies for states around the obstacle.

To get the optimal policies, policy iteration or value iteration can be used to improve the current policy.

Policy Iteration

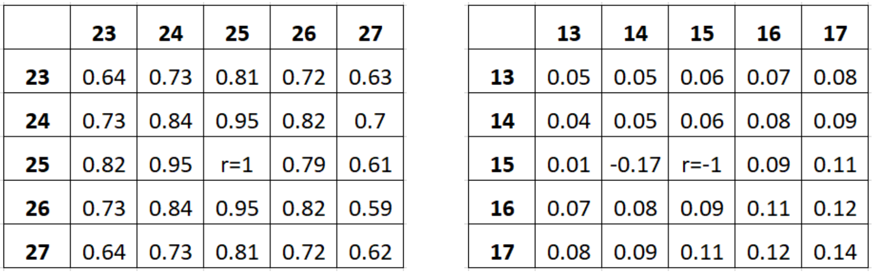
When policy evaluation is done, not only the values of all states are calculated but also the action values for all states based on (2.10). The policy improvement theorem states that, let π and $\pi^{'}$ be any pair of deterministic policies such that,

$$q_\pi\left(s,\pi^{'}(s)\right)\geq v_\pi(s)\,, \tag{2.14}$$

for all $s\in\mathcal{S}$. Then the policy $\pi^{'}$ must be as good as, or better than, π. So, a policy iteration sequence can be formed:

$$\pi_0 \overset{e}{\to} v_{\pi_0} \overset{i}{\to} \pi_1 \overset{e}{\to} v_{\pi_1} \overset{i}{\to} \pi_2 \overset{e}{\to} v_{\pi_2} \overset{i}{\to} \dots \overset{i}{\to} \pi_* \overset{e}{\to} v_{\pi_*}, \tag{2.15}$$

where $\overset{e}{\to}$ denotes a policy evaluation and $\overset{i}{\to}$ denotes a policy improvement. Following the policy iteration algorithm in [1], the state values are initialized with 0, and the probabilities of action selections are $\{right, up, left, down\} = \{0.1, 0.2, 0.3, 0.4\}$ for all $s\in\mathcal{S}$. Figure 2.5 shows the optimal values around the goal and the obstacle. To illustrate the optimal policies for all states, the action values are to be calculated. And Figure 2.6(a) shows the optimal policies for all non-terminal states, where each non-terminal state is divided into four parts indicating four actions, and the blue triangle is to indicate the maximum action value for a state, i.e. the optimal policy for a state.



	23	24	25	26	27
23	0.64	0.73	0.81	0.72	0.63
24	0.73	0.84	0.95	0.82	0.7
25	0.82	0.95	r=1	0.79	0.61
26	0.73	0.84	0.95	0.82	0.59
27	0.64	0.73	0.81	0.72	0.62

(a) Around the goal.

	13	14	15	16	17
13	0.05	0.05	0.06	0.07	0.08
14	0.04	0.05	0.06	0.08	0.09
15	0.01	-0.17	r=-1	0.09	0.11
16	0.07	0.08	0.09	0.11	0.12
17	0.08	0.09	0.11	0.12	0.14

(b) Around the obstacle.

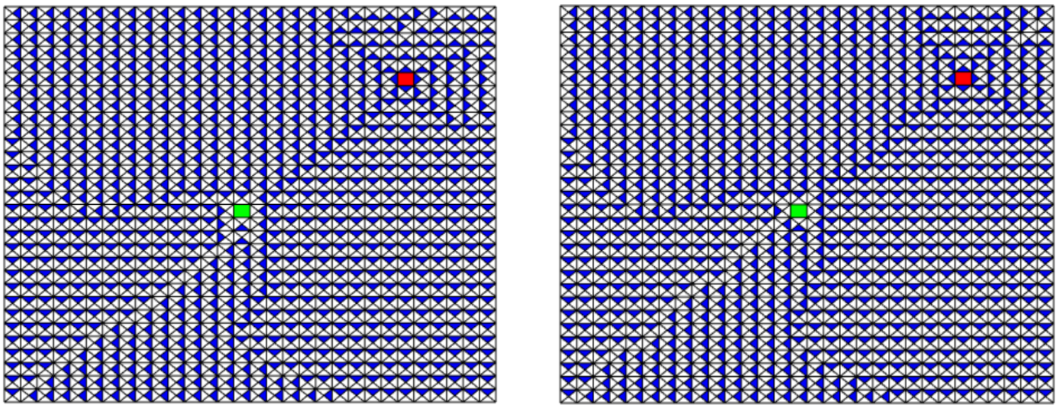
FIGURE 2.5: Optimal values from policy iteration.

Value Iteration

With policy iteration, the policy needs to be improved after a complete evaluation of current policy, which, per se, is an iterative computation taking a long time. However, the policy improvement and policy evaluation can be combined together, and one special case is value iteration[1], of which the update is defined by:

$$v_{k+1}(s) = \max_a \sum_{s^{'},r} p\left(s^{'}, r \mid s, a\right) \left[r + \gamma v_k\left(s^{'}\right)\right], \tag{2.16}$$

for all $s \in \mathcal{S}$. In this way, the same results can be achieved efficiently, which is evidenced by the experimental result shown in Figure 2.6(b). Compared with policy iteration, which takes 13.2 seconds, value iteration only uses 4.9 seconds to find this policy.


(a) Policy iteration.

(b) Value iteration.

Figure 2.6: Optimal policies from dynamic programming.

2.3.2 Monte Carlo Methods

From here, the given dynamics of the defined environment will be ignored and the optimal policies are to be learned by trial-and-error interaction with the environment. To estimate the state values:

$$v_{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right], \tag{2.17}$$

a sample return can be used to replace the expected return, which leads to the Monte Carlo methods. Without the dynamics, state values are not sufficient to suggest policies, instead, action values need to be estimated. Monte Carlo methods can estimate values based on averaging sample returns:

$$\begin{aligned} q_n(s,a) &= \frac{\text{sum of cumulative rewards after } a \text{ taken in } s}{\text{number of times } a \text{ taken in } s \text{ up to episode } n} \\ &= \frac{G_1 + G_2 + \cdots + G_{n-1}}{n-1}, \end{aligned} \tag{2.18}$$

which will also converge to the expected value when the number of samples goes to infinite.

Besides, the estimate for one state is not based on others, as in the case in dynamic programming, which gives the advantage of always starting the episodes from the states of interest, estimating based on these states while ignoring others. But, to find the optimal values, every state-action pair should be visited, i.e. the policy for decision-making should be exploratory. There are two ways: on-policy and off-policy, used to ensure exploration. On-policy means that the policy used for generating the episode is to be evaluated and improved, whereas inside off-policy methods, there are two policies: one for selecting actions and another one is the target policy to be improved. One of such kind of policies that ensures both convergence and exploration is ϵ-soft policy:

$$\pi\left(a, \mid S_t\right) \leftarrow \begin{cases} 1-\epsilon+\epsilon/\left|\mathcal{A}\left(S_t\right)\right| & \text{if } a = A^* \\ \epsilon/\left|\mathcal{A}\left(S_t\right)\right| & \text{if } a \neq A^* \end{cases} \tag{2.19}$$

for all $a \in \mathcal{A}\left(S_t\right)$, $A^* \leftarrow \arg\max_a Q\left(S_t, a\right)$, with ties broken arbitrarily, and the parameter ϵ can be either made arbitrarily large to ensure exploration or small to ensure exploitation and drive the convergence to a an optimal policy.

On-Policy Monte Carlo

Here, the ϵ-soft policy is to be used to maintain the probabilities for all actions in a state, and then one action will be chosen according to the probability distribution. When one episode ends with a final return, the action values are to be updated:

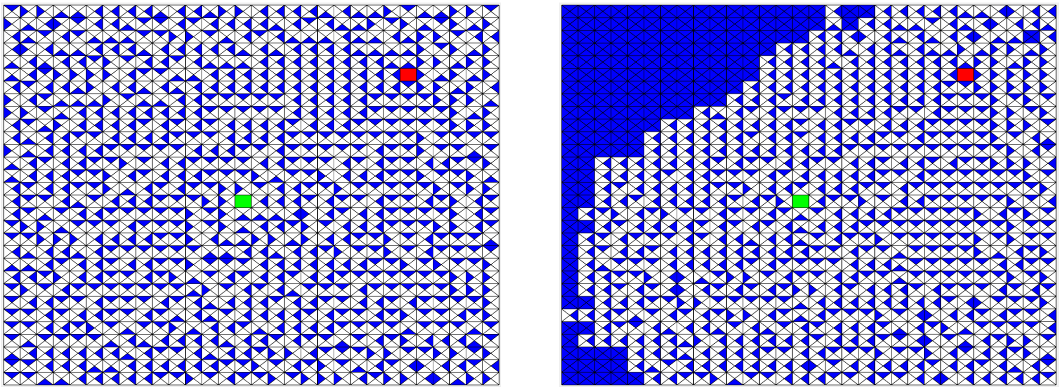
$$q(S_t, A_t) = \frac{\sum_{i=1}^{n} Returns(S_t, A_t)}{n}, \tag{2.20}$$

where the $Returns(S_t, A_t)$ is the accumulated reward in a episode when A_t is selected in S_t, and n is the number of episodes where S_t appears and A_t happens to be chosen. Meanwhile, the policy will be changed if the maximum action value for a state is held by a new action.

With the algorithm provided in [1], the implementation starts the iteration with $\epsilon = 0.8$ and slowly decrease it to 0.1. Final policy is shown in Figure 2.7(a).

Off-Policy Monte Carlo

The on-policy method learns action values that are near-optimal since it has to keep exploring with probability ϵ, whereas off-policy provide us the opportunity to approach optimal policy by using two policies: one for selecting actions and another one is learned to become the optimal policy. Another part of this algorithm provided in [1] that is different from the on-policy method is that an incremental implementation for updating



(a) On-policy.

(b) Off-policy.

FIGURE 2.7: Policies from Monte Carlo methods.

the action values is chosen:

$$\begin{aligned} q_n(s,a) &= \frac{1}{n-1}\sum_{i=1}^{n-1} G_{n-1} \\ &= \frac{1}{n-1}\left(G_{n-1} + \sum_{i=1}^{n-2} G_{n-2}\right) \\ &= \frac{1}{n-1}\left(G_{n-1} + (n-2)\frac{1}{n-2}\sum_{i=1}^{n-2} G_{n-2}\right) \\ &= \frac{1}{n-1}\left(G_{n-1} + (n-2)\, q_{n-1}(s,a)\right) \\ &= \frac{1}{n-1}\left(G_{n-1} + (n-1)\, q_{n-1}(s,a) - q_{n-1}(s,a)\right) \\ &= q_{n-1}(s,a) + \frac{1}{n-1}\left[G_{n-1} - q_{n-1}(s,a)\right], \end{aligned} \tag{2.21}$$

which, for example for $n = 2$, obtaining $q_2(s,a) = G_1$ for arbitrary $q_1(s,a)$. The general form of this update rule is:

$$NewEstimate = OldEstimate + StepSize\left[Target - OldEstimate\right], \tag{2.22}$$

where the expression $[Target - OldEstimate]$ is considered to be an *error* in the estimate, and the *StepSize* is also called learning rate, denoted by α or, α_t for general case. The advantage of this way is that all the previous returns do not need to be buffered as in the on-policy method where the value is estimated based on averaging sample returns.

In the implementation, an ϵ-soft policy is used for generating the behavior while a greedy policy is to be learned, and the ϵ decreases slowly from 0.8 to 0.3. The obtained policy is shown in Figure 2.7(b).

The agent starts exploring from state (5,5), and after 100 000 episodes, as shown in the

Figure 2.7, the learned policies mainly suggest how to find a good way form (5,5) to the goal state (30,30), especially in Figure 2.7(b), which is one of the features of Monte Carlo methods that they aim to evaluate and improve policies for the states of interest while ignoring the others.

2.3.3 Temporal-Difference Learning

In dynamic programming, the value of current state is updated based on the estimate of the next state; in Monte Carlo methods, action value is updated based on sample returns. Here, with temporal-difference learning, the update is based on samples of the estimate of next state:

$$q\left(S_t, A_t\right) = q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma q\left(S_{t+1}, A_{t+1}\right) - q\left(S_t, A_t\right)\right], \tag{2.23}$$

for $A_{t+1} \in \mathcal{A}\left(S_{t+1}\right)$, which will have the advantages of both dynamic programming and Monte Carlo methods: it is not needed to wait until the end of the episode to get the return and the values can be learned from experience. The update method in (2.23), based on only current state and next state, is called TD(0) and it is the simplest version of TD methods. Similar to Monte Carlo methods, to keep exploration to find the optimal policies,there are also on-policy and off-policy methods for TD(0).

On-Policy TD(0)

As shown in (2.23), the method relies on information about the variables S_t, A_t, R_{t+1}, S_{t+1}, and A_{t+1}, which explains why the algorithm is also called sarsa, introduced in [1]. On-policy means that the policy used to select actions is also the policy used for evaluation and improvement. So, (2.23) can be more precisely expressed as:

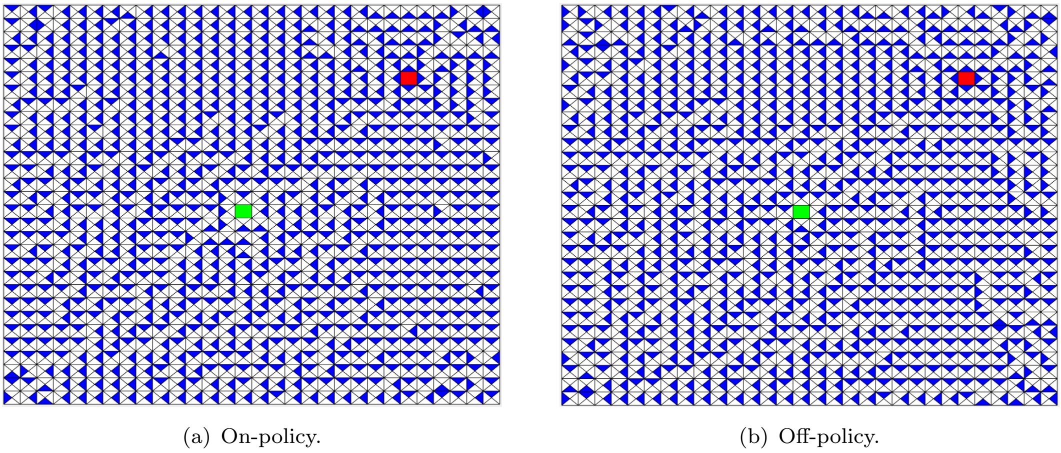
$$q_\pi\left(S_t, A_t\right) = q_\pi\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma q_\pi\left(S_{t+1}, A_{t+1}\right) - q_\pi\left(S_t, A_t\right)\right], \tag{2.24}$$

where A_{t+1} is determined by the current policy π. In the implementation, an ϵ-soft policy is used with the ϵ decreasing slowly from 0.8 to 0.3, and the result is shown in Figure 2.8(a).

Off-Policy TD(0)

Compared with on-policy method, an exploratory policy should be used here for generating the actions while the policy used for evaluation and improvement can be greedy:

$$\begin{aligned} q\left(S_t, A_t\right) &= q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) - q\left(S_t, A_t\right)\right] \\ &= q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_{a \in \mathcal{A}\left(S_{t+1}\right)} q\left(S_{t+1}, a\right) - q\left(S_t, A_t\right)\right], \end{aligned} \tag{2.25}$$



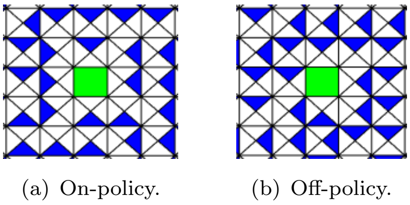
(a) On-policy.

(b) Off-policy.

FIGURE 2.8: Policies from Temporal-Difference Learning.

where the max operator shows the greediness. In the implementation, the same ϵ-soft policy as in on-policy method is used, and the result is shown in Figure 2.8(b).

Compared with the results of Monte Carlo methods shown in Figure 2.7, it can be observed that the optimal policies for most of the states are found here since the update of the action value is based on successor state, like the case in dynamic programming. For off-policy learning, it assumes that the next action is selected according to the optimal policy. With this greediness, it normally finds the shortest way to reach the goal while it can also hit the obstacle since the policy for generating actions is not optimal. The policies around the obstacle are highlighted in Figure 2.9. For safety, the on-policy method will take a detour to make sure that the agent will not hit the obstacle while off-policy aims to find the shortest path.



(a) On-policy.

(b) Off-policy.

FIGURE 2.9: Policies for states around the obstacle.

2.3.4 Linear Function Approximation

The above methods all have one or more value tables to store related values for each state so that the policy for that state can then be calculated. However, when there are many states in the environment, it will not be practical to apply these methods not only because of the memory needed for large tables, but the time and data needed to fill them accurately [1]. To scale it to practical problems, function approximation methods can be applied to estimate state values or action values. In such a way, only the features that can represent states need to be found, then the state value can be approximated, for example, by:

$$\begin{aligned} v_{\omega}\left(s\right) &= \omega_0 \cdot 1 + \omega_1\phi_1\left(s\right) + \omega_2\phi_2\left(s\right) + \cdots + \omega_m\phi_m\left(s\right) \\ &= \boldsymbol{\omega}^\top \boldsymbol{\phi}\left(s\right), \end{aligned} \tag{2.26}$$

where $\boldsymbol{\phi}\left(s\right)$ denotes the features and $\boldsymbol{\omega}$ is the parameter vector that needed to be found.

In the implementation, four features $(\phi_1, \phi_2, \phi_3, \phi_4)$ for the self-defined environment are extracted: the horizontal and vertical distances from the the agent to goal and obstacle. For example, the agent is at location (20,20), so the horizontal and vertical distances to the goal are 5 and 5, while the horizontal and vertical distances to the obstacle are -5 and -5.

The objective of this methods is to minimize the *Mean Squared Value Error*, denoted by [1]:

$$\overline{VE}\left(\boldsymbol{\omega}\right) = \sum_{s\in\mathcal{S}} \mu\left(s\right)\left[v_\pi\left(s\right) - v_{\boldsymbol{\omega}}\left(s\right)\right]^2, \tag{2.27}$$

where $\mu\left(s\right)$ is the state distribution, representing how much the error between the true value $v_\pi\left(s\right)$ and the approximate value $v_{\boldsymbol{\omega}}\left(s\right)$ in the state s matters. The states are assumed to appear with the same distribution, and with *Stochastic gradient-descent* (SGD) methods, the error can be reduced by adjusting the weight in the gradient descent direction:

$$\begin{aligned} \boldsymbol{\omega}_{t+1} &= \boldsymbol{\omega}_t - \frac{1}{2}\alpha\nabla\left[v_\pi\left(S_t\right) - v_{\boldsymbol{\omega}_t}\left(S_t\right)\right]^2 \\ &= \boldsymbol{\omega}_t + \alpha\left[v_\pi\left(S_t\right) - v_{\boldsymbol{\omega}_t}\left(S_t\right)\right]\nabla v_{\boldsymbol{\omega}_t}\left(S_t\right) \\ &= \boldsymbol{\omega}_t + \alpha\left[v_\pi\left(S_t\right) - v_{\boldsymbol{\omega}_t}\left(S_t\right)\right]\boldsymbol{\phi}\left(S_t\right), \end{aligned} \tag{2.28}$$

where α is a positive step-size parameter. And similar to TD methods, the true state value $v_\pi\left(S_t\right)$ can be approximated using $R_{t+1} + \gamma v_{\boldsymbol{\omega}_t}\left(S_{t+1}\right)$, which yields the general SGD method for the parameter update:

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \alpha\left[R_{t+1} + \gamma v_{\boldsymbol{\omega}_t}\left(S_{t+1}\right) - v_{\boldsymbol{\omega}_t}\left(S_t\right)\right]\boldsymbol{\phi}\left(S_t\right). \tag{2.29}$$

However, to determine the policy for each state, the action values need to be learned:

$$q_{\boldsymbol{\omega}}\left(s, a\right) = \omega_0 \cdot 1 + \omega_1\phi_1\left(s, a\right) + \omega_2\phi_2\left(s, a\right) + \cdots + \omega_m\phi_m\left(s, a\right), \tag{2.30}$$

and the features for state-action pair can then be encoded by [22]:

$$\boldsymbol{\phi}(s) = \begin{bmatrix} \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \\ \phi_4(s) \end{bmatrix} \Rightarrow \boldsymbol{\phi}(s, a_1) = \begin{bmatrix} \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \\ \phi_4(s) \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{16\times 1}, \cdots, \boldsymbol{\phi}(s, a_4) = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \\ \phi_4(s) \end{bmatrix}_{16\times 1}. \tag{2.31}$$

Here, sarsa with linear function approximation [1] is implemented and after training for 100 000 episodes, another 200 episodes are tested, and the distribution of the number of steps for reaching the goal in each episode is shown in Figure 2.10, where the x-axis indicates the number of steps in an episode and y-axis indicates the distributions.

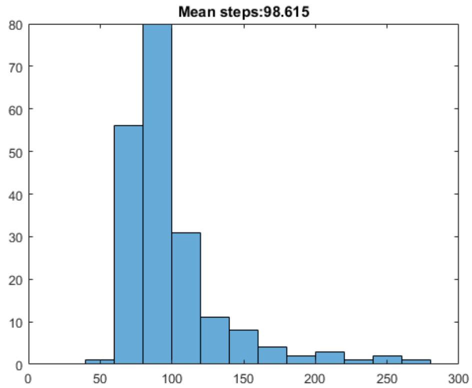


FIGURE 2.10: Sarsa with linear function approximation: the number of steps for reaching the goal.

With the learned parameters, the size of the environment is changed from 30 × 30 to 300 × 300 and the agent can still find the goal while avoiding the obstacle, which shows the scalability of this reinforcement learning method to practical problems with large state space.

2.3.5 Deep-Q Network

Linear function approximation seems to have a great scalability, yet in many situations, it might even not be appropriate to use linear function approximator. Neural networks are a popular method used as a non-linear function approximator. With hidden layers between input features and output values, some features that are not observable can also be obtained. Figure 2.11 shows a typical architecture for a deep neural network with two hidden layers in between. where each hidden layer consists of a linear function

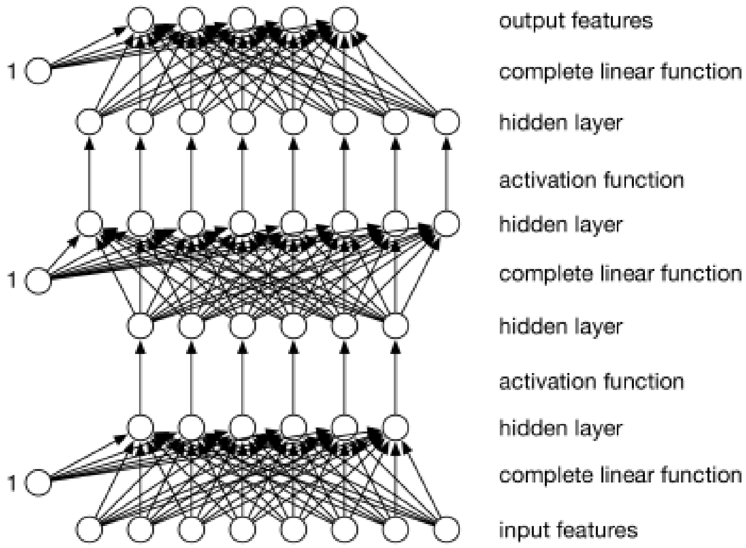


FIGURE 2.11: A deep neural network [2].

and an activation function. Linear function is similar to what is shown in Section 2.3.4, so the input value of a hidden unit is:

$$h\_in = \boldsymbol{\omega}^\top \boldsymbol{v}, \tag{2.32}$$

where the $\boldsymbol{v}$ is the feature vector if the hidden unit is in the first hidden layer; otherwise it is the output vector of the preceding hidden layer. The activation function is to scale the output values of hidden layers using, for example, *sigmoid* function:

$$h\_out = \frac{1}{1 + e^{-h\_in}}, \tag{2.33}$$

of which, the input values can be limited in the range $(0, 1)$, which can accelerate the computation speed so as to speed up learning process.

The features are extracted referring the method mentioned in [23], and instead of using pixels, three matrices are used to represent where the goal, obstacle, and agent are. And the weight normalization method presented in [24] is used to initialize the parameters so as to accelerate the learning process. For updating the parameters, back-propagation is used, referring the step-by-step examples mentioned in [25, 26] .

In the final implementation, two hidden layers with 40 units for each are applied, and instead of getting features for each action, as done in 2.3.4, the features here is to represent the state while there are four units at output layer representing each action value. And for stability, experience replay is introduced to the implementation: there is a 30000-long buffer for storing transition information, and when one episode is done, 3000 instances will randomly be sampled from the buffer and used to update the weights.

After training, 200 extra episodes is tested, and the distribution of the number of steps for reaching the goal in each episode is shown in Figure 2.12. Compared with 100 000 episodes of training in 2.3.4, here it is only 20 000 episodes and yet it has a better performance, which may be the reason as mentioned before, with hidden layers, it can find the features that are not observable or easy to be extracted. However, to get it trained only for 20 000 episodes, it takes around 37 hours. Deep-Q Network is a useful tool, but it is not necessary to apply it to a simple task as defined here.

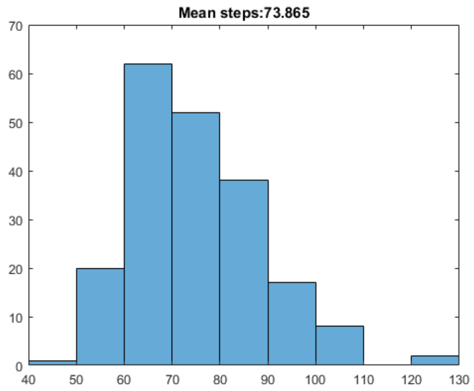


FIGURE 2.12: Deep-Q network: the number of steps for reaching the goal.

2.3.6 Policy Gradient with Linear Function Approximation

Instead of computing a value function, a stochastic policy can also be approximated directly using an independent function approximator with its own parameters [27]. With parameter vector $\boldsymbol{\theta}$, the policy can be written as $\pi\left(a \mid s, \boldsymbol{\theta}\right) = Pr\left\{A_t = a \mid S_t = s, \boldsymbol{\theta_t} = \boldsymbol{\theta}\right\}$, indicating the probability of taking action a given the state s and parameter vector $\boldsymbol{\theta}$ at time t. To calculate the $\pi\left(a \mid s, \boldsymbol{\theta}\right)$, here, parameterized numerical preferences $h\left(s, a, \boldsymbol{\theta}\right)$ can be formed for each state-action pair:

$$h\left(s, a, \boldsymbol{\theta}\right) = \boldsymbol{\theta}^{\top} \boldsymbol{\phi}\left(s, a\right), \tag{2.34}$$

where $\boldsymbol{\phi}\left(s, a\right)$ is the feature vector constructed the same as in Section 2.3.4. Then the actions can be assigned with the probabilities of being selected in each state according to their preferences, for example, using an exponential soft-max distribution method:

$$\pi\left(a \mid s, \boldsymbol{\theta}\right) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}. \tag{2.35}$$

Following a certain policy, the agent can interact with the environment to get a trajectory of states, actions, and returns, where the returns are the sum of rewards from time step

t onwards, G_t. The learning of $\boldsymbol{\theta}$ is based on the gradient ascent of the performance measure $J(\boldsymbol{\theta}) = \mathbb{E}[G_1 \mid \boldsymbol{\theta}]$ of that policy, i.e. maximizing the performance:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t), \tag{2.36}$$

where α is a positive step size or learning rate, and gradient of performance with respect to the policy parameter [1] is provided by *policy gradient theorem*:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla_{\boldsymbol{\theta}} \pi(a \mid S_t, \boldsymbol{\theta}) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a \mid S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a \mid S_t, \boldsymbol{\theta})}{\pi(a \mid S_t, \boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}) \right].
\end{aligned} \tag{2.37}$$

where the final term in the bracket can be sampled, which is also equal to the gradient, and the update rule can then be written as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}). \tag{2.38}$$

And for the case with linear function approximation, $\nabla_{\boldsymbol{\theta}} \ln \pi(a \mid s, \boldsymbol{\theta})$ is calculated by:

$$\nabla_{\boldsymbol{\theta}} \ln \pi(a \mid s, \boldsymbol{\theta}) = \boldsymbol{\phi}(s,a) - \sum_b \pi(b \mid s, \boldsymbol{\theta}) \boldsymbol{\phi}(s,b). \tag{2.39}$$

To reduce the variance [1], a baseline can be included in the equation:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha (G_t - b(S_t)) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}), \tag{2.40}$$

and one natural choice for this baseline is an estimate of the state value, $v_{\boldsymbol{\omega}}(S_t)$, and the return G_t can also be expressed by:

$$G_t = R_{t+1} + \gamma v_{\boldsymbol{\omega}}(S_{t+1}), \tag{2.41}$$

which leads to the method called actor-critic (AC), and the parameters are updated by:

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha (R_{t+1} + \gamma v_{\boldsymbol{\omega}}(S_{t+1}) - v_{\boldsymbol{\omega}}(S_t)) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}) \\
&= \boldsymbol{\theta}_t + \alpha \delta_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}),
\end{aligned} \tag{2.42}$$

where the $\boldsymbol{\omega}$ is updated using (2.29) mentioned in 2.3.4.

In the implementation, the features used to approximate policy are extracted using the same way as in 2.3.4, while the features for estimating state values contains two items: the absolute distance from the agent to goal and obstacle. After being trained for 100 000 episodes, another 200 episodes are tested, and the distribution of the number of steps for each episode is shown in Figure 2.13. Compared with the results from sarsa with linear function approximation (Figure 2.10) and deep-Q network (Figure 2.12), the policy gradient method has a much better performance in both average number of steps and stability.

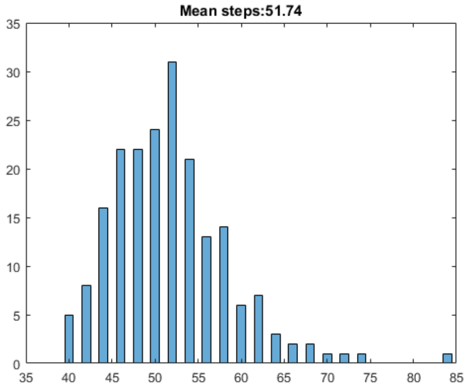


FIGURE 2.13: Policy gradient with linear function approximation: the number of steps for reaching the goal.

2.4 Summary

From the implementations and evaluations for the above algorithms, it can be noticed that: if the dynamics of the environment are given, then DP methods could be good approaches to solve the problem. The other methods are mostly used for environments where the model is unknown. Monte Carlo methods are good for the situation where there are many states and only some of them are of interest, while TD methods are good in the sense that it is not needed to wait until the end of an episode to update all values. And the on-policy methods, compared with off-policy, can find a safer way to the goal since they do not assume that the actions taken after current state are all going to be optimal.

To scale it to practical problems where there exists a huge state space and action space, function approximation methods can then be applied, with which, the values will be estimated based on old experience where the features are similar to the current state.

Linear approximator is easy to understand and to implement while non-linear approximator could be better for piratical problems where the value cannot be approximated in a linear way. Instead of finding the values and then calculate the policies, policy gradient methods can also be used to search for policies directly, and the policy learned will be either deterministic or stochastic, depending on the environment.

For the coordinated multi-agent task that mentioned in Section 1.1, the model for the environment is unknown and the state space is way larger than the experimental environment, a good idea is to choose methods with function approximation. Given that non-linear approximators are suitable for complex problems, so, for simplicity, linear function approximation related methods can be introduced to solve the task.

Chapter 3

Multi-Agent Reinforcement Learning

In this chapter, formal models for multi-agent reinforcement learning (MARL) are presented and also the approaches for solving multi-agent task. The chapter begins with the introductions to the mathematical models. And with a self-defined multi-agent simulation environment, different methods are discussed and some solutions implemented in MATLAB are presented. The code can be accessed through the link: `https://goo.gl/8REkDz`.

3.1 Mathematical Models

The MARL framework is also based on the previous model shown in Figure 2.1. The difference is that the actions of other agents also have effects on the environment, which normally leads to different result from the same action for an agent. Multi-agent framework follows the assumptions [2]: the agents can act autonomously, each with its own observations of the environment and the other agents; the outcome depends on the actions of all the agents; each agent has its own utility that depends on the outcome; and agents act to maximize their own utility. To model such a framework, the focus has turned to game theory, which is indeed the study of multiple interacting agents trying to maximize their rewards [5], especially the model of Matrix Games (MGs) and Stochastic Games (SGs).

Matrix Games

An N-player MG can be defined as a tuple $(n, \mathcal{A}_{1\ldots n}, \mathcal{R}_{1\ldots n})$, where n is the number of agents, $\mathcal{A}_i$ is the available action space for agent i ($i \in [1, n]$), and $\mathcal{R}_i$ is the reward of agent i. A strategy π for agent i is the probability distribution of possible action $a_i \in \mathcal{A}_i$, it can be either pure strategy with probability 1 for a certain action or mixed strategy by choosing actions stochastically. The optimality here is to find the best response strategy for an agent, where the best means the agent can not expect a higher reward if it changes the strategy. The collection of response strategies from all agents is a Nash Equilibrium, and for MGs, there is at least one Nash Equilibrium [5]. A payoff matrix is always along with a MG, which is the reward of chosen joint actions. Table 3.1 shows an example of Rock-Paper-Scissors, which is a *two-person matrix game*. And the row index is specified with actions of the first player while the column for second. Rewards are specified with left hand indicating first player and right hand for second. However, the payoff matrix is hard to define in advance when applying repeated matrix games to MARL [5, 13].

TABLE 3.1: Rock-Paper-Scissors reward matrix [5].

	Rock	**Paper**	**Scissors**
Rock	(0, 0)	(-1, 1)	(1, -1)
Paper	(1, -1)	(0, 0)	(-1, 1)
Scissors	(-1, 1)	(1, -1)	(0, 0)

Stochastic Games

Currently, MARL has focused on SGs. An N-player SG can be defined as a tuple $(n, \mathcal{S}, \mathcal{A}_{1\ldots n}, T, \mathcal{R}_{1\ldots n})$, where n is the number of agents, $\mathcal{S}$ is the state set, $\mathcal{A}_i$ is the available action space for agent i ($i \in [1, n]$), T is a transition function which depends on the actions of all players, and $\mathcal{R}_i$ is the reward of player i. The collection of strategies is called *joint policy*, denoted by $\pi =< \pi_i, \pi_{-i} >$, where π_{-i} refers to the joint policy for all players except for i. Unlike single-agent framework, the joint actions of agents result in next state and rewards [5]. Yet, the goal of an agent in a SG is still to maximize its expected reward. If the rewards for all agents are the same, then it is a fully cooperative game.

3.2 Multi-Agent Task Description

The environment for multi-agent experiments is defined in MATLAB as shown in Figure 3.1: inside an area of size 30×30, two agents are located at $(7, 7)$ and $(7, 21)$ while there are two goal locations $(14, 14)$ and $(21, 14)$. The agents are moving so that they can reach both goals at the same time. Different locations for both agents represent the state of the environment, and the terminal states are defined by: when both agents are at the same location, the interaction is terminated with a reward $r = -100$; when each agent is occupying one goal (distance is less than 2), the interaction is terminated with a reward $r = 10$. Apart from the terminal states, the agent will get a reward $r = -1$ each time they interact with the environment. There are four possible actions in each state for each agent, $\mathcal{A} = \{up, down, right, left\}$, which will deterministically take the agent to the direction, except that the heading is taking the agent out of the area, in which case the agent will keep the location unchanged.

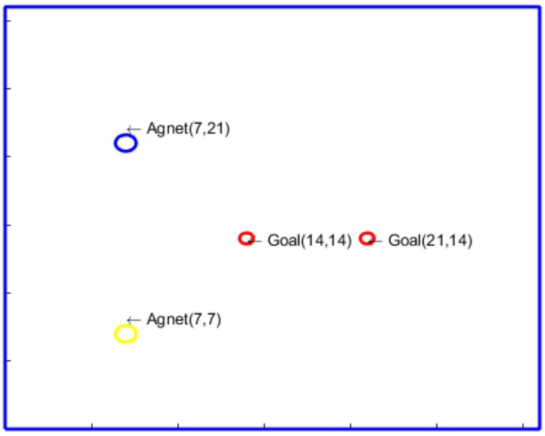


Figure 3.1: Multi-agent environment defined in MATLAB.

3.3 Solutions for Multi-Agent Environment

Similar to single agent environment, the value function for agent i of a state s, following the joint policy π, is defined by:

$$\begin{aligned} v_{\pi}^{i}(s) &= \mathbb{E}_{\pi}\left[G_{t}^{i} \mid S_{t}=s\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+k+1}^{i}+\gamma G_{t+1}^{i} \mid S_{t}=s\right] \\ &= \sum_{a} \pi(a \mid s) \sum_{s^{'}} T\left(s^{'} \mid s, a\right)\left[r^{i}\left(s^{'} \mid s, a\right)+\gamma v_{\pi}^{i}\left(s^{'}\right)\right], \end{aligned} \tag{3.1}$$

where $\pi(a \mid s)$ is the probability of choosing joint action a in state s. Even that the state value is defined for each agent, the expected return depends on the joint policy. And the value of taking the joint action a in a state s and thereafter following joint policy π is defined by:

$$q_{\pi}^{i}(s, a) = \sum_{s^{'}} T\left(s^{'} \mid s, a\right)\left[r^{i}\left(s^{'} \mid s, a\right)+\gamma v_{\pi}^{i}\left(s^{'}\right)\right]. \tag{3.2}$$

For updating the action values, an incremental implementation can be applied:

$$\begin{aligned} q_n^i(s,a) &= q_{n-1}^i(s,a) + \alpha\left[G_{n-1}^i - q_{n-1}^i(s,a)\right] \\ &= q_{n-1}^i(s,a) + \alpha\left[r_{n-1}^i(s^{'} \mid s,a) + \gamma q_{n-1}^i\left(s',a'\right) - q_{n-1}^i(s,a)\right], \end{aligned} \tag{3.3}$$

where the update can either be based on the final return or a sample of a transition.

Efforts have been put into the research of the methods for solving multi-agent learning, and the proposed approaches integrate developments in the areas of single agent RL, game theory, and direct policy search techniques [11]. In [28], a comparison among basic Q-learning algorithms was presented. With centralized Q-learning:

$$q(s,a_1,...,a_n) \leftarrow q(s,a_1,...,a_n) + \alpha\left[r + \gamma \max_{a'_1,\ldots,a'_n} q(s',a'_1,...,a'_n) - q(s,a_1,...,a_n)\right], \tag{3.4}$$

a good performance is achieved while more information is needed, and it also needs to maintain a larger state-action space. Then with decentralized Q-learning algorithm:

$$q_i(s,a_i) \leftarrow q_i(s,a_i) + \alpha\left[r + \gamma \max_{a'} q_i(s',a') - q_i(s,a_i)\right], \tag{3.5}$$

the state-action space is reduced. However, the agent can get punished even it takes the right action because other agents may take wrong actions, and the joint action leads to punishment. To avoid this, distributed Q-learning can be used, in which, Q values can only be incremented. The key issue with distributed Q-learning algorithm is that optimistic agents do not manage to achieve the coordination between multiple optimal joint actions. So, hysteretic Q-learning was proposed:

$$\delta \leftarrow r + \gamma \max_{a'} q_i(s',a') - q_i(s,a_i), \tag{3.6}$$

$$q_i(s,a_i) \leftarrow \begin{cases} q_i(s,a_i) + \alpha\delta & \text{if } \delta \geq 0 \\ q_i(s,a_i) + \beta\delta & \text{else} \end{cases}, \tag{3.7}$$

where α and β are the increase and decrease rates of Q-values. This learning method is decentralized in the sense that each agent builds its own Q-table whose size is independent of the agents' number and linear in function of its own actions. According to the experiments, the hysteretic Q-learning algorithm performs as well as a centralized algorithm while using a much smaller Q-value table. In [12], a comparison between single agent Q-learning and team Q-learning was presented. The reason single agent Q-learning can work in multi-agent setting is that the agent can take other agents' actions as a random disturbance or random noise in the environment. As for team Q-learning algorithm, it was specifically designed to solve the purely cooperative stochastic game problem. However, the result showed that, single agent Q-learning can sometimes outperform team Q-learning. To improve the team Q-learning method, it was also pointed out that the probability of random actions should be increased to overcome the local maximum problem and the number of training iterations should also be incremented so that the robot can explore enough states and converge to the optimal policy. However, the methods above, using a lookup table, are only feasible when the state and action spaces of the system are small. To cope with the exponential growth of the problem size with increasing number of agents, [29] showed an approach for multi-agent deep reinforcement learning (MADRL), where problem representation and training techniques were discussed.

Apart from adapting Q-learning to multi-agent settings, policy gradient methods have also been applied, especially actor-critic method [18, 30, 31]. To ease the training, the framework of centralized training with decentralized execution is adopted, i.e. the critic is provided with extra information, for example, the policies of other agents, while the actor only uses the local observation to choose actions. And for a fully cooperative environment, there can only be one critic for all actors since they will always have the same reward, whereas, for a mixed cooperative-competitive environment, it is needed to have one critic for each actor.

As discussed in Section 2.4, linear appoximators are to be used for approximating the values and policies. And the following part will present the implementations of centralized Q-learning, hysteretic Q-learning, and multi-agent actor-critic (MAAC) with linear function approximation.

3.3.1 Centralized Q-learning

For the defined environment, in a state s, the action a consists of (a_1, a_2) from two agents, where $a_1, a_2 \in \mathcal{A}$. And the size of the action a is therefore $4 \times 4 = 16$. The features for a state is extracted as $\boldsymbol{\phi}(s) = (\phi_1, \phi_2, ..., \phi_{10})$, and the descriptions for all the features is shown in Table 3.2. Similar to what is mentioned in Section 2.3.4, the features for state-action pair will be of size 160, and the action values can then be approximated by:

$$q_{\boldsymbol{\omega}}(s, a_1, a_2) = \omega_0 \cdot 1 + \omega_1 \phi_1(s, a_1, a_2) + \omega_2 \phi_2(s, a_1, a_2) + \cdots + \omega_{160} \phi_{160}(s, a_1, a_2), \quad (3.8)$$

where, $\boldsymbol{\omega}$ is the parameter vector to be learned, and it is updated by:

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \alpha \left[r + \gamma \max_{a_1', \ldots, a_n'} q_{\boldsymbol{\omega}_t}(s', a_1', a_2') - q_{\boldsymbol{\omega}_t}(s, a_1, a_2) \right] \boldsymbol{\phi}(s, a_1, a_2), \quad (3.9)$$

TABLE 3.2: Description of the features for a state.

Feature	Description
ϕ_1	horizontal distances from the **first** agent to the **first** goal
ϕ_2	vertical distances from the **first** agent to the **first** goal
ϕ_3	horizontal distances from the **first** agent to the **second** goal
ϕ_4	vertical distances from the **first** agent to the **second** goal
ϕ_5	horizontal distances from the **second** agent to the **first** goal
ϕ_6	vertical distances from the **second** agent to the **first** goal
ϕ_7	horizontal distances from the **second** agent to the **second** goal
ϕ_8	vertical distances from the **second** agent to the **second** goal
ϕ_9	horizontal distances from the **first** agent to the **second** agent
ϕ_{10}	vertical distances from the **first** agent to the **second** agent

where s, a_1, and a_2 are the state and actions chosen at time step t, while r, s', a_1', and a_2' are the reward, state, and available actions at time step $t+1$.

In the implementation, the learning rate α is set to be 0.05, and the discount factor γ is set to be 0.99. To keep exploration, an ϵ-soft policy (see Section 2.3.2) is used with ϵ decreasing slowly from 0.8 to 0.1. After 5000 episodes, the sum of rewards during each episode is collected, and the mean values and standard deviations for every 200 episodes are calculated, as shown in Figure 3.2. The mean values are connected with a line while the error bar shows standard deviations.

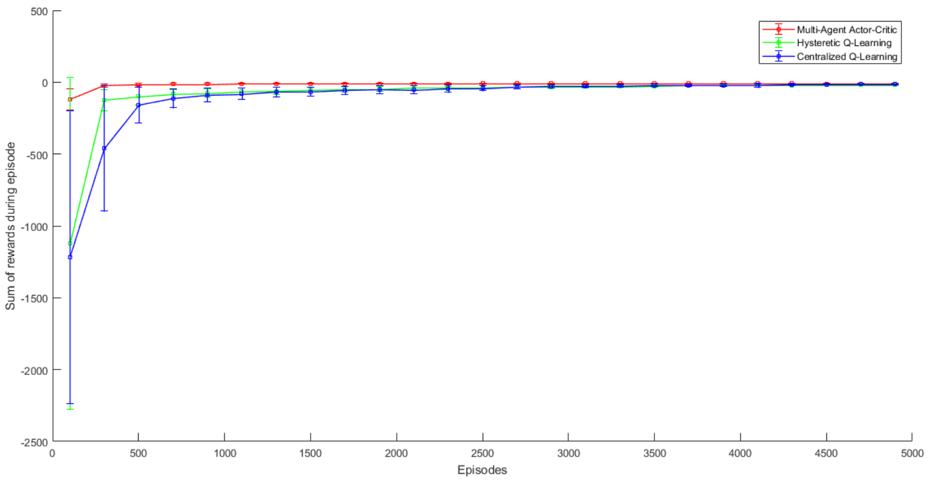


FIGURE 3.2: Mean values and standard deviations.

3.3.2 Hysteretic Q-learning

Hysteretic Q-learning provides the ability to control the agents in a decentralized manner. In a certain state of the above-defined environment, the action space for an agent is only of size 4, which shrinks down the size of the features representing the state-action pair from 160 (feature size of centralized Q-learning) to 40. And to approximate the action values for two agents, two parameter vectors $\boldsymbol{\omega_1}, \boldsymbol{\omega_2}$ are needed and to be learned. The update rule for the parameter vectors is similar to the equations addressed in (3.6) and (3.7):

$$\delta \leftarrow r + \gamma \max_{a'} q_i(s', a') - q_i(s, a_i), \tag{3.10}$$

$$\boldsymbol{\omega}_{i,t+1} \leftarrow \begin{cases} \boldsymbol{\omega}_{i,t} + \alpha\delta & \text{if } \delta \geq 0 \\ \boldsymbol{\omega}_{i,t} + \beta\delta & \text{else} \end{cases}, \tag{3.11}$$

where i indicates the index of agent, $a_i, a' \in \mathcal{A}$, α and β are the increasing rate and decreasing rate for updating the parameters.

In the implementation, the increasing rate α is set to be 0.05, the decreasing rate β is set to be 0.005, and the rest hyperparameters are the same as that in centralized Q-learning. The reward information is also shown in Figure 3.2.

3.3.3 Multi-Agent Actor-Critic

A centralized critic is to be learned to critique the actors. The critic approximates the values for state with a parameter vector $\boldsymbol{\omega}$ based on features $\boldsymbol{\phi}(s) = (\phi_1, \phi_2, \phi_3)$, where ϕ_1 is the distance from the first agent to a goal and ϕ_2 is the distance from the second agent to another goal, which makes the sum of these two distances smaller, and ϕ_3 is the distance from the first agent to the second agent. So, the state value is approximated by:

$$v_\omega(s) = \omega_0 \cdot 1 + \omega_1\phi_1(s) + \omega_2\phi_2(s) + \omega_3\phi_3(s)\,. \tag{3.12}$$

For each actor, to calculate the $\pi_i(a_i \mid s)$, parameterized numerical preferences $h_i(s, a_i, \boldsymbol{\theta}_i)$ need to be formed for each state-action pair:

$$h_i(s, a_i, \boldsymbol{\theta}) = \boldsymbol{\theta}_i^\top \boldsymbol{\phi}(s, a_i)\,, \tag{3.13}$$

where i indicates the index of agent, $a_i \in \mathcal{A}$, $\boldsymbol{\theta}_i$ is the parameter vector for parametrizing the policy, and $\boldsymbol{\phi}(s, a_i)$ is constructed based on the features listed in Table 3.2. When an interaction between agents and environment happens, the parameter vectors are updated by:

$$\delta_t = r + \gamma v_{\boldsymbol{\omega}_t}\left(s'\right) - v_{\boldsymbol{\omega}_t}\left(s\right), \tag{3.14}$$

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \alpha\delta_t\boldsymbol{\phi}\left(s\right), \tag{3.15}$$

$$\boldsymbol{\theta}_{i,t+1} = \boldsymbol{\theta}_{i,t} + \alpha\delta_t\nabla_{\boldsymbol{\theta}_i}\ln\pi\left(a_i \mid s, \boldsymbol{\theta}_i\right), \tag{3.16}$$

where s and a_i are the state and action chosen at time step t, while r and s' are the reward and state at time step $t+1$.

In the implementation, the learning rate for updating actor parameter vectors is set to be 0.0025 while that for updating critic parameter vector is set to be 0.05. And the discount factor γ and the number of training episodes are the same as above mentioned. Figure 3.2 shows the rewards statistics.

From the Figure 3.2, it can be observed that centralized Q-learning learns the parameters slower, and the reason may be that it has a rather larger action space. Hysteretic Q-learning has a good performance while MAAC can solve the task even faster.

3.4 Summary

It can be observed that, for centralized Q-learning, if there are many agents, the action space is going to increase exponentially. And with the trend of decentralized control, hysteretic Q-learning and MAAC are more practical. However, for MAAC, at least one centralized critic (usually one critic for each actor) is needed, which means, during execution, the parameters will not be able to be updated. And this will only work when the dynamics of the environment are stable. And for the coordinated multi-agent task that mentioned in Section 1.1, the dynamics will not change. So, to learn the parameters faster, MAAC should be chosen.

Chapter 4

Agent-Based Systems

In this chapter, fundamentals of software agents, agent-based systems, and related technologies are presented. The chapter first discusses the properties for agents in general and then different approaches for designing the agent-based systems. Finally, the implementations for the benchmark model mentioned in Section 1.1 is shown.

4.1 General Software Agents

An agent is a computational entity that is capable of autonomous action on behalf of its user or owner to solve a problem or reach a goal. The set of features of an agent can be listed as follows [21]:

- *Autonomy:* Internal state and its goals can only be changed by itself.
- *Responsiveness:* It perceives the environment by using sensors and responds by driving the actuators.
- *Proactiveness:* It has the ability to proactively anticipate the possible changes and react to them.
- *Goal-orientation:* It takes the action which can drive itself to the goal.
- *Smart behavior:* It can be equipped with the representation of the part of the environment that it acts in and thus, it has the knowledge of that area and behaves smartly.
- *Social ability:* A complex system usually consists of many agents interacting with each other, they need to communicate and even negotiate one another.

- *Learning capabilities:* Instead of being constantly told what to do, it has to learn to achieve the autonomy.

To achieve the smartness of an agent, two types of agents are defined: deliberative and reactive agents. For a deliberative agent, it has an observation of at least part of the environment so that with the reasoning architecture and interaction history, it can make a decision proactively. While for a reactive agent, the decision will be made based entirely on the present. For example, for a deliberative robot, if there is an obstacle in front, it will take the action which will not make it bump into the obstacle according to the interaction history. Whereas for a reactive robot, it might be programmed as: turn right when it hits an obstacle. The need for social ability always arises in MASs where there are many agents aim to work cooperatively or competitively. In such a case, the agent needs a means for communication.

4.2 The Foundation for Intelligent Physical Agents

To develop a comprehensive system, especially needed for industrial applications, standards always play the most important role. In the agent community, the IEEE Foundation for Intelligent Physical Agents (FIPA) is the key standardization body. FIPA was established as an international non-profit association to develop a collection of standards relating to software agent technology, and during the evolution [3], many agent-related ideas and software tools have been proposed and developed.

4.2.1 Agent Communication

Agents are computational entities distributed in a system, which can be modeled as components and connectors, and components are consumers, producers and mediators of communication messages exchanged via connectors [3]. The most commonly used communication language standard is Agent Communication Language (ACL), a FIPA standard, which is based on performative or communicative acts such as agree, propose, refuse, request, and query-if [21]. A FIPA-ACL message contains parameters needed for effective communication such as sender, receiver, and content. Among all the parameters, the only mandatory one is performative, according to which, FIPA has defined a set of interaction protocols such as Request Protocol, Cancel-Meta Protocol, and Query Protocol. Figure 4.1 shows the sequential diagram for the FIPA Request Protocol. The *Initiator* wants to request the participant to perform a certain action by sending a message using *request* as performative. Then the *Participant* processes the received request and decide whether to accept or refuse it. If it is refused, the *Initiator* will receive a

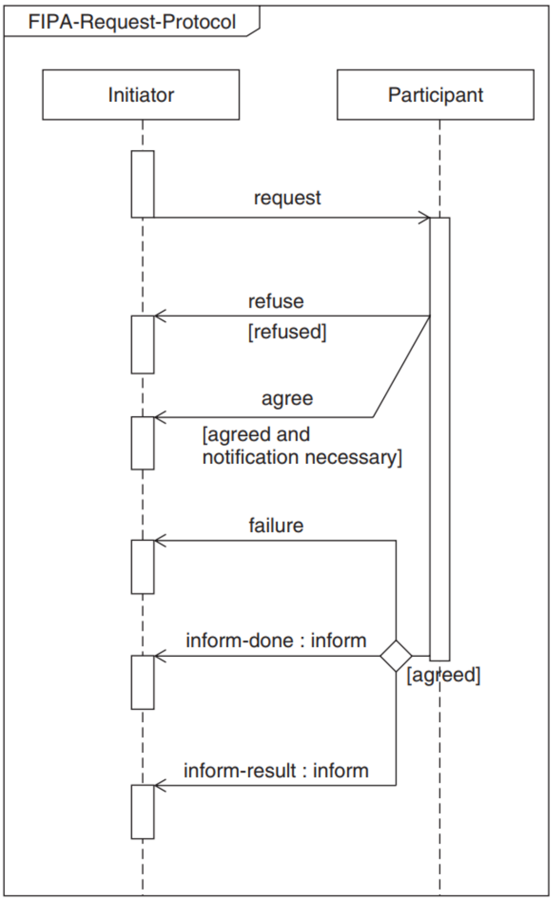


FIGURE 4.1: The FIPA Request Interaction Protocol [3].

message with *refuse* as performative and end this communication. Otherwise, the *Initiator* will receive *agree*, and further, the *Participant* will execute the requested action and notify the *Initiator* the result.

4.2.2 Agent Management

Apart from agent communication, agent management is also one of the fundamental aspects of agent-based systems addressed in FIPA. It is a framework within which the logical reference model for all operations related to agents can be established such as creation, registration, and communication. Figure 4.2 shows such a framework. Agent Platform (AP) provides the physical infrastructure for deploying agents and it consists of Agent Management System (AMS), which manages the operation of an AP such as the creation and deletion of agents, and Directory Facilitator (DF), which provides yellow pages services to all agents. AMS contains descriptions for agents that in the AP, and DF contains descriptions of the services. Message Transport Service (MTS) is for transporting FIPA-ACL messages mentioned above.

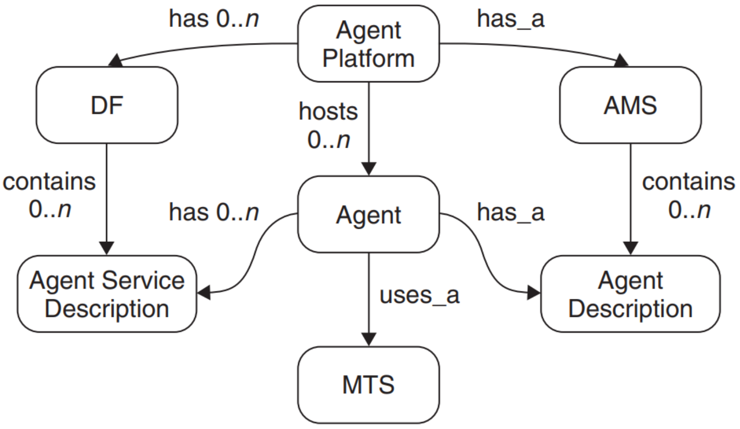


FIGURE 4.2: Agent management ontology [3].

4.2.3 Java Agent Development Platform

JADE is a software framework for developing FIPA compliant agent-based systems. The abstraction of a software agent is implemented in Java, which provides a simple and friendly API [20]. Figure 4.3 shows a JADE distributed architecture. The AP can be divided into many containers situated in different hosts, yet there is only one main container, which holds the AMS and DF. One or more agents can be located in any container. And agents can communicate transparently regardless of whether they are part of the same container or even the same platform.

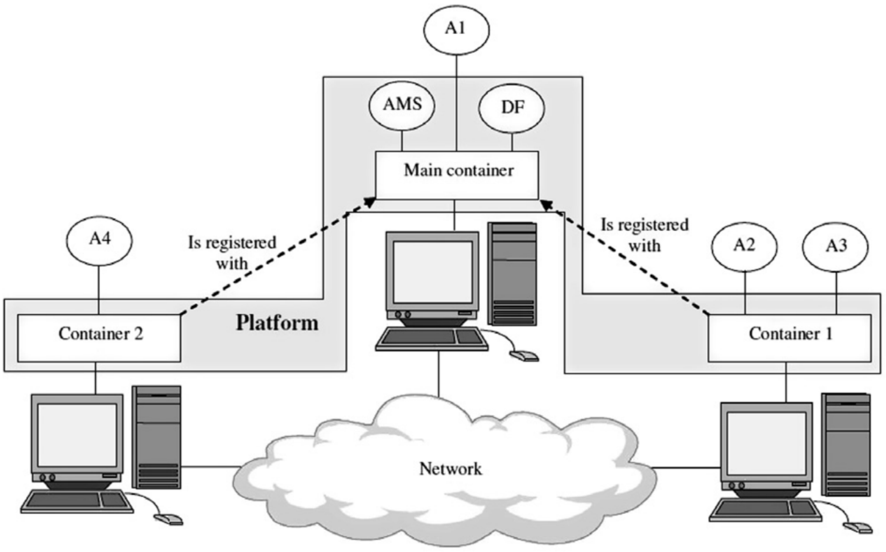


FIGURE 4.3: JADE distributed architecture [4].

4.3 The Design of Multi-Agent Systems

Agent-based systems have been explored in a wide range of domains. And the characteristics such as autonomy, social ability, and proactiveness of MAS have been identified as core ingredients for the reliability [21]. For developing a MAS, there are two ways of designing the architecture: coupled design and embedded design.

4.3.1 Coupled Design

Coupled design is currently popular in automation scenarios, and a coupled design for the benchmark model is shown in Figure 4.4. Agents are situated in the agent space, where the characteristics such as social abilities for agents are implemented. Whereas the control part like actuators is located in system space. For the benchmark, there are three agents in the agent space, and the broker agent communicates with the robot agent to ask for a service. In the system space, one Sphero robot is controlled by one Raspberry Pi, and the Raspberry Pi will need information from the agent space to decide the movement of the Sphero robot. With such a design, a middle-ware technology is normally needed for handling the communication between the agent space and system space, for instance, Middleware-based LabMap using publisher/subscriber architecture [32], a product of CBB Software Company in Germany, can be applied here.

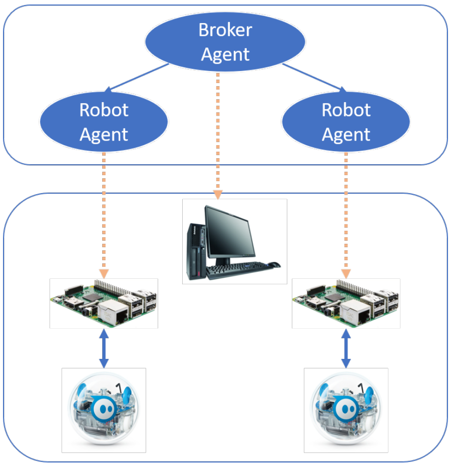


FIGURE 4.4: A coupled design.

4.3.2 Embedded Design

An embedded design for the benchmark model is shown in Figure 4.5. In this case, the agent usually has access to native control operations of the system. With the proper architectural support, the embedded design promotes the decoupling of agents logically

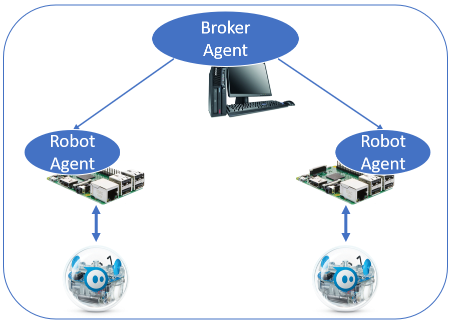


FIGURE 4.5: An embedded design.

and geographically, effectively enabling the creation of plug-and-produce entities comprising the artifact being controlled, the controller, and the agent [21].

Because of the advantages of embedded design and also the fact that it has been rarely explored in industrial scenarios, the final architecture for the benchmark model is developed based on embedded design.

4.4 Final Design for Benchmark Model

The structural diagram for the final design is shown in Figure 4.6. The *Sphero* is controlled by a *Raspberry Pi*, in which, one *Robot Agent* is embedded. To observe the locations of *Spheros*, the camera is driven by the *Camera Vision*, an application on *PC*. And the location information will be transferred to *Robot Agents* in *Raspberry Pi* through *UDP Broadcasting*. The *Broker Agent* embedded in PC takes the user input as commands and then communicates with *Robot Agents* to finish the task.

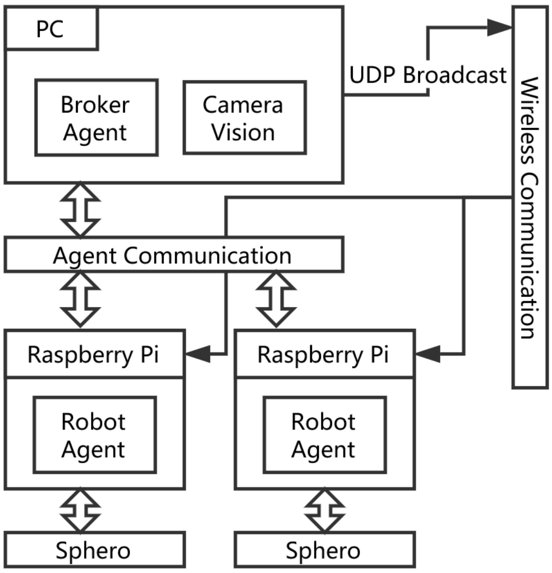


FIGURE 4.6: Structural diagram.

Chapter 5

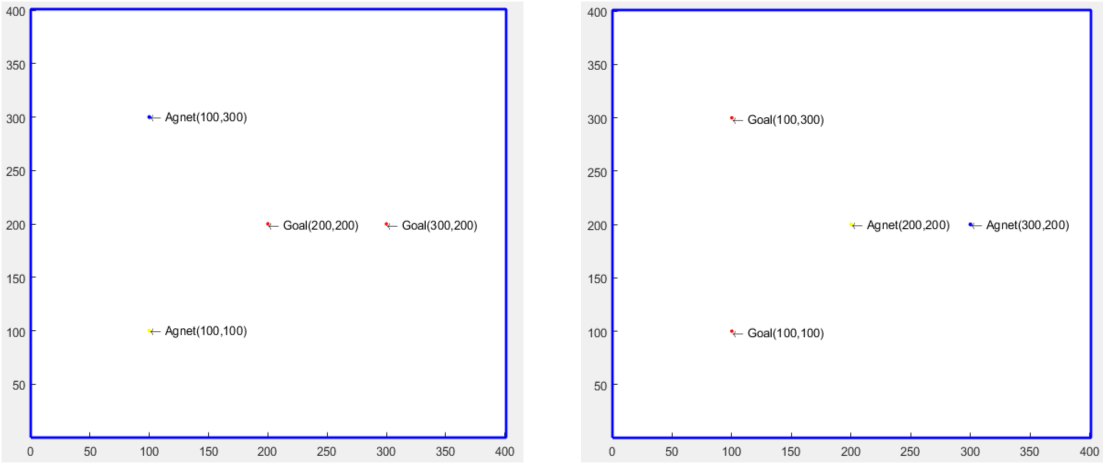
Implementation and Experiment

In this chapter, a simplified model for the benchmark is set up in MATLAB. The parameter vectors used to control the Spheros are obtained from the MATLAB simulation. Then the whole system is implemented based on JADE platform, and the parameters learned from simulated experience are applied to controlling the Spheros.

The MATLAB program and the system implementation can be found through the link: `https://goo.gl/DNvauV`. Finally, the experiment is carried out and the result is presented.

5.1 Robot Agents Training

As described in the Section 1.1, the two Spheros need to occupy the two landmarks first and then roll back to the stations. To train the robot agents with simulated experience, two models are built in MATLAB: the goal locations are that of landmarks and the goal locations are that of stations. As shown in Figure 5.1: inside an area of size 400×400, two stations are located at $(100, 100)$ and $(100, 300)$ while there are two landmarks located at $(200, 200)$ and $(300, 200)$. Since the Sphero has a certain size, the terminal states are empirically defined as: when agents are at locations with distance less than 10, the interaction is terminated with a reward $r = -1000$; when each agent is occupying one goal (distance is less than 5), the interaction is terminated with a reward $r = 100$. And when a Sphero takes an action, for example *up*, it will move by 4 units in that direction while the location will stay unchanged if the action takes the agent out of the defined area. As mentioned in Section 3.4, MAAC is to be used here. And there is one difference compared with the implementation in Section 3.3.3 that two Spheros have random start locations instead of always starting from the same so that they can be originally situated



(a) Spheros roll toward the landmarks.

(b) Spheros roll toward the stations.

FIGURE 5.1: Models for simulation.

at any positions inside the area. After training, the parameter vectors for policies are obtained, as shown in the Table 5.1, which will be used in the agent-based system to control the Spheros.

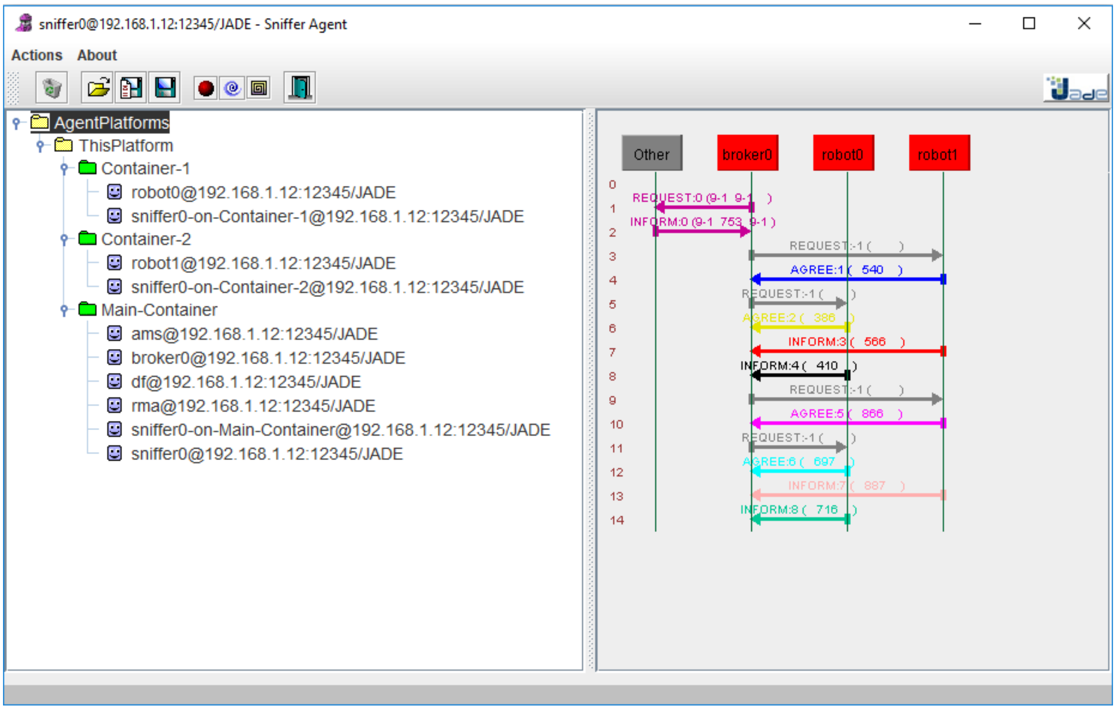
TABLE 5.1: Learned Parameters for Benchmark Model.

Parameter	**Move to Landmarks**		**Move to Stations**	
Index	**First Agent**	**Second Agent**	**First Agent**	**Second Agent**
1	0.46	0.78	0.69	1.00
2	0.99	1.13	1.10	0.71
3	4.01	2.58	6.59	1.06
4	0.21	0.50	-0.07	0.34
5	6.98	3.40	4.76	0.94
6	0.40	1.68	0.27	1.22
7	2.69	4.60	1.32	5.00
8	0.84	0.63	0.44	-0.42
9	2.92	7.69	1.32	5.99
10	0.50	1.60	0.78	0.81
11	-3.17	5.42	-0.83	0.72
12	4.64	1.94	3.90	1.50
13	0.37	1.01	0.98	0.94
14	7.59	1.21	4.93	0.88
15	-0.92	0.34	2.18	0.64
16	0.74	6.39	2.60	3.41
17	0.47	0.12	0.25	1.42

18	1.48	6.62	0.26	5.61
19	-1.23	-0.48	1.43	0.54
20	-1.84	2.17	-4.00	5.49
21	0.62	-0.58	1.99	0.70
22	1.22	0.21	2.16	0.78
23	-3.47	-2.20	-5.18	-1.41
24	0.56	0.36	0.45	0.95
25	-2.46	0.10	-5.35	-0.74
26	1.33	-0.55	-0.10	1.50
27	-0.67	-4.01	-0.07	-5.30
28	0.70	1.54	1.69	-0.40
29	1.92	-3.18	-0.29	-5.27
30	1.52	1.33	0.22	0.18
31	6.22	-4.19	2.04	-1.89
32	-4.66	-1.35	-4.94	-0.81
33	0.66	0.69	-0.63	1.08
34	-6.15	-0.59	-2.52	-0.02
35	-1.22	-1.33	0.02	1.53
36	-0.22	-5.75	-0.31	-4.01
37	-0.55	1.09	0.43	0.41
38	-1.02	-6.14	-1.10	-2.58
39	-0.95	-1.81	0.37	1.73
40	1.88	-2.71	5.05	-3.36
41	-0.59	1.37	-1.39	1.09

5.2 Multi-Agent System Implementation

The project is created in NetBeans integrated development environment (IDE). As mentioned in Section 4.4, there should be three agents: one broker agent and two robot agents. The broker agent knows the requirement and requests the robot agents to reach the goal. And the broker agent is in the main container situated on the computer while the robot agents are in two containers situated on two Raspberry Pi. Figure 5.2 shows an example of how the agents communicates. The broker sends a FIPA compliant *RE-*



sniffer0@192.168.1.12:12345/JADE - Sniffer Agent

- AgentPlatforms
  - ThisPlatform
    - Container-1
      - robot0@192.168.1.12:12345/JADE
      - sniffer0-on-Container-1@192.168.1.12:12345/JADE
    - Container-2
      - robot1@192.168.1.12:12345/JADE
      - sniffer0-on-Container-2@192.168.1.12:12345/JADE
    - Main-Container
      - ams@192.168.1.12:12345/JADE
      - broker0@192.168.1.12:12345/JADE
      - df@192.168.1.12:12345/JADE
      - rma@192.168.1.12:12345/JADE
      - sniffer0-on-Main-Container@192.168.1.12:12345/JADE
      - sniffer0@192.168.1.12:12345/JADE

FIGURE 5.2: Agent communication.

QUEST message to each robot agent and also receives *AGREE* from them. Then robot agents will move to the landmarks and send *INFORM* to broker agent when they occupy both landmarks.

To control the movement of the robot agents, a state machine is to be used here, as shown in Figure 5.3. There are five states for the robot agents: *IDLE*, *TO_LOCATION*,

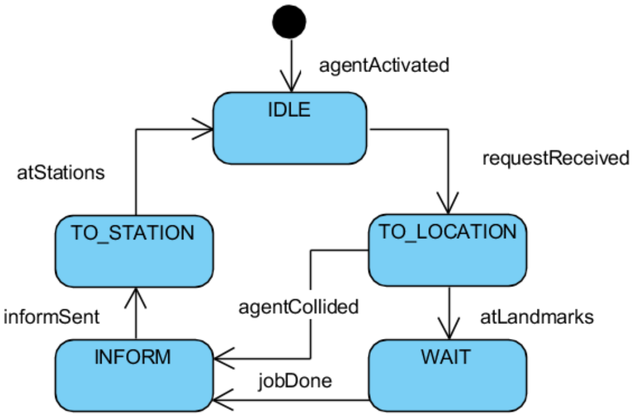


FIGURE 5.3: Robot agents control.

WAIT, *INFORM*, and *TO_STATION*. When the robot agent is activated, it will be in state *IDLE*, where the agent is standing by, waiting for a request. When a request is initiated by the broker agent, it sends back *AGREE*, and goes into state *TO_LOCATION*, meanwhile, it needs to initialize the goal locations and the weights for predicting the state-action values, and the Java code is as follows:

```
/************** begin of java code *****************/
this.state = MoveState.TO_LOCATION;//set the state to TO_LOCATION
this.env.set_goal_to_target();//set the goal locations to landmarks
this.myPE.set_weights_for_target();// set the weights used for landmarks
/**************  end of java code  *****************/
```

Listing 5.1: Initialization before in state *TO_LOCATION*

When the robot agent is in the state *TO_LOCATION*, it moves, coordinating with another robot agent, to occupy both landmarks, and the Java code is as follows:

```
/************** begin of java code *****************/
this.agent_locations = this.myAgent.get_agent_locations();//get current
    locations from UDP

this.status = this.env.set_current_location(this.agent_locations);//check if
    landmarks occupied or agents collided
if(this.status == 1){
   this.state = MoveState.WAIT;
}
else{
    if(this.status == -1){
         this.state = MoveState.INFORM;
      }
}
this.probabilities = this.myPE.predict(this.agent_locations);//predict the
    probabilities for actions
this.action_idx = this.randsample(this.env.actionSpace,
    this.probabilities);//random sample an action
this.mySphero.move(this.hrad[this.action_idx]);//execute the action
this.myAgent.doWait(50);
this.mySphero.stop();
/**************  end of java code  *****************/
```

Listing 5.2: Spheros control

If they collide with each other during movement, they will fail the task and end up in state *INFORM*, where they are supposed to inform the broker agent that they failed the task. Otherwise, they end up at both landmarks and wait for seconds. Then they inform the broker agent that the task is fulfilled. In the state *INFORM*, since the next state is *TO_STATION*, the goal locations and weights need to be set again, and the Java code is as follows:

```
/************** begin of java code *****************/
this.state = MoveState.TO_STATION;//set the state to TO_LOCATION
this.env.set_goal_to_station();//set the goal locations to stations
this.myPE.set_weights_for_station();// set the weights used for stations
/**************  end of java code  *****************/
```

Listing 5.3: Initialization before in state *TO_STATION*

Finally, they move back to stations, also coordinating with each other, and become idle again when they reach both stations. The code for this state is similar to that in Listing 5.2.

5.3 Experiment

A video is recorder to show the Spheros moving controlled by RL method, and it can be viewed through the link: https://goo.gl/U1A8yN.

There are two parts in the video. The first part is showing the single agent RL integrated into the agent-based system, and the purpose of this is only to get familiar with integrating RL to an agent-based system. While the second part is the result for the benchmark model, in which, it can be clearly observed that when the two robot agents are requested to occupy the landmarks, they can move, coordinating with each other, to the landmarks. When the task is fulfilled, they can then also move back to stations, still coordinating with each other.

Chapter 6

Discussion and Future Work

In this chapter, conclusions of the work done in this thesis is presented. With the knowledge obtained during conducting this project, directions for future work is also provided.

6.1 Conclusion

Research and work have been done in MARL and agent-based systems, based on which, this thesis focuses on integrating a RL method into an agent-based system.

Chapter 2 investigates and evaluates RL methods for single agent, and the result shows that the methods with function approximation, compared with the methods without it, are better in the situation where the state space and the action space are very lager since they need less storage room for only storing the parameter vectors and they can choose the actions based on old experience where the features are similar to the current state. Then Chapter 3 presents the implementations for centralized Q-learning, hysteretic Q-learning, and MAAC with linear function approximation, and evidenced by the result shown in Figure 3.2, MAAC can find the solution effectively and efficiently.

Chapter 4 presents a detailed introduction to concepts related to agent and agent-based systems. For developing an agent-based system, the JADE platform is also briefly discussed. Since the advantages that embedded design has, for example facilitating the decoupling of agents logically and geographically, it is chosen as the design for the defined benchmark model in Section 1.1.

Chapter 5 first presents a simplified model for the defined problem in Section 1.1 and trains it with the MAAC method to get the parameter vectors used to select actions.

Then an agent-based system is implemented, applying these parameter vectors to control the Spheros. With the experiment conducted and the result shown in the video (https://goo.gl/U1A8yN), it can be observed that the defined cooperative navigation problem in Section 1.1 is solved by integrating the MAAC method into the agent-based system developed with JADE platform. Moreover, the parameter vectors are obtained through simulations based on a simplified model for the defined problem, which indicates that parameters learned from simulated experience can be used in a real control system.

6.2 Future Work

With the theories for the methods using linear function approximation, the solutions for most industrial applications, especially where it involves the control of multi-robot, can be easily found. However, the hardest part is to integrate the RL methods into the agent-based system because they need accurate sensory input to describe the state of the environment so that the action values can be precisely predicted. In the thesis, the state of the environment is defined as the locations of the Spheros, and to get the locations, a Simulink project is implemented where a camera is used to track the Spheros. It works when there are only two Spheros rolling in a low speed as the case in this thesis, but it can be predicted that it will not work properly when there are more Spheros or they need to respond much faster. Research can be done for finding an appropriate way to represent the state of the real environment.

Learning with simulated experience will greatly shorten the time needed to train the agent. However, a simplified model built in the simulation platform mostly cannot represent the real environment and the interaction among the agents and the environment, especially when the dynamics of the environment change with time. In this case, training with simulated experience while executing in the real environment will not work. So, the next step is to train the agents when they are situated in the real environment. And probably, a balance could be found between the simulated and real experience so that not only the training time can be shortened but the agents will be able to learn continuously.

Bibliography

[1] Richard S. Sutton, Andrew G. Barto, and Harry Klopf. *Reinforcement Learning : An Introduction Second edition*. MIT Press, 2018.

[2] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, New York, NY, USA, 2nd edition, 2017. ISBN 110719539X, 9781107195394.

[3] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. 2007. ISBN 9780470057476. doi: 10.1002/9780470058411.

[4] Maha A. Metawei, Salma A. Ghoneim, Sahar M. Haggag, and Salwa M. Nassar. Load balancing in distributed multi-agent computing systems. *Ain Shams Engineering Journal*, 3(3):237–249, 2012. ISSN 20904479. doi: 10.1016/j.asej.2012.03.001.

[5] Gonçalo Neto. From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods Learning Theory Course.

[6] Katia Sycara, Anandeep Pannu, Mike Williamson, Dajun Zeng, and Keith Decker. Distributed intelligent agents. *IEEE Expert-Intelligent Systems and their Applications*, 11(6):36–46, 1996. ISSN 08859000. doi: 10.1109/64.546581.

[7] Jing Xie and Chen-Ching Liu. Multi-agent systems and their applications. *Journal of International Council on Electrical Engineering*, 7(1):188–197, 2017. ISSN 2234-8972. doi: 10.1080/22348972.2017.1348890. URL https://www.tandfonline.com/doi/full/10.1080/22348972.2017.1348890.

[8] Michael Wooldridge. *An Introduction to MultiAgent Systems [Paperback]*. 2009. ISBN 0470519460. doi: 10.1108/eb050656. URL https://www.google.com/books?hl=hr{&}lr={&}id=X3ZQ7yeDn2IC{&}oi=fnd{&}pg=PR13{&}dq=Wooldridge,+M.,+An+introduction+to+multiagent+systems.+2009:+John+Wiley+{%}26+Sons{&}ots=WFoevw8u54{&}sig=pW{_}SoLDnMqc6fkrAnJzFJIv8phI{%}5Cnhttp://www.amazon.com/Introduction-MultiAgent-Syste.

[9] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. ISSN 14764687. doi: 10.1038/nature24270.

[10] Peter Stone and Manuela Veloso. Multiagent systems: a survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000. ISSN 09295593. doi: 10.1023/A:1008942012299.

[11] L. Busoniu, R. Babuska, B De Schutter, and B. De Schutter. A comprehensive survey of multiagent reinforcement learning. *Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(2):156–172, 2008. ISSN 1094-6977. doi: 10.1109/TSMCC.2007.913919.

[12] Ying Wang and Clarence W. De Silva. Multi-robot box-pushing: Single-agent Q-learning vs. team Q-learning. In *IEEE International Conference on Intelligent Robots and Systems*, pages 3694–3699, 2006. ISBN 142440259X. doi: 10.1109/IROS.2006.281729.

[13] Erfu Yang and Dongbing Gu. Multiagent reinforcement learning for multi-robot systems: A survey. *University of Essex Technical Report CSM-404, . . .*, pages 1–23, 2004. doi: 10.1.1.2.1602. URL http://computerscience.nl/docs/vakken/aibop/MAS-reinforcement-learning.pdf{%}5Cnhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.1602{&}rep=rep1{&}type=pdf{%}5Cnpapers2://publication/uuid/7E7FE019-FB17-4EB1-82E1-B90BE22E856A.

[14] M. A Wiering. Multi-agent reinforcement learning for traffic light control. *In Machine Learning: Proceedings of the Seventeenth International Conference*, pages pp. 1151–1158, 2000.

[15] Bram Bakker, Shimon Whiteson, Leon Kester, and Frans C a Groen. Traffic light control by multiagent reinforcement learning systems. *Interactive Collaborative Information Systems*, pages 475–510, 2010. ISSN 1860949X. doi: 10.1007/978-3-642-11688-9_18.

[16] Leo Raju, Sibi Sankar, and R. S. Milton. Distributed optimization of solar micro-grid using multi agent reinforcement learning. In *Procedia Computer Science*, volume 46, pages 231–239, 2015. ISBN 9781479939756. doi: 10.1016/j.procs.2015.02.016.

[17] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. JADE: A software framework for developing multi-agent applications. Lessons learned. *Information and Software Technology*, 50(1-2):10–21, 2008. ISSN 09505849. doi: 10.1016/j.infsof.2007.10.008.

[18] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Neural Information Processing Systems (NIPS)*, 2017.

[19] Michael Hviid Aarestrup. Control sphero with cpp code. https://github.com/miaar7/Sphero, 2018.

[20] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - A FIPA - compliant agent framework. *Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 99:97–108, 1999. ISSN 00380644. doi: 10.1145/375735.376120. URL http://www.dia.fi.upm.es/{~}phernan/AgentesInteligentes/referencias/bellifemine99.pdf.

[21] Paulo Leitão and Stamatis Karnouskos. *Industrial Agents: Emerging Applications of Software Agents in Industry.* 2015. ISBN 9780128004111. doi: 10.1016/C2013-0-15269-5.

[22] Alborz Geramifard, Thomas J. Walsh, Stefanie Tellex, Girish Chowdhary, Nicholas Roy, and Jonathan P. How. A tutorial on linear function approximators for dynamic programming and reinforcement learning. *Foundations and Trends® in Machine Learning*, 6(4):375–451, 2013. ISSN 1935-8237. doi: 10.1561/2200000042. URL http://dx.doi.org/10.1561/2200000042.

[23] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative Multi-Agent Control Using Deep Reinforcement Learning. *Adaptive Learning Agents (ALA) 2017*, 2017. URL http://ala2017.it.nuigalway.ie/papers/ALA2017{_}Gupta.pdf.

[24] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *NIPS*, 2016.

[25] Matt Mazur. A Step by Step Backpropagation Example – Matt Mazur. URL https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[26] Josh Patterson. Introduction To Deep Q-Learning – Josh Patterson – Medium. URL https://medium.com/@joshpatterson{_}5192/introduction-to-deep-q-learning-1bded90a6193.

[27] Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. Technical report, 1999.

[28] Laëtitia Matignon, Guillaume J. Laurent, and Nadine Le Fort-Piat. Hysteretic Q-Learning : An algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *IEEE International Conference on Intelligent Robots and Systems*, pages 64–69, 2007. ISBN 1424409128. doi: 10.1109/IROS.2007.4399095.

[29] Maxim Egorov. Multi-Agent Deep Reinforcement Learning. *CS231n*, page 8, 2016.

[30] Chun-Gui Li, Meng Wang, and Qing-Neng Yuan. A Multi-agent Reinforcement Learning using Actor-Critic methods. In *2008 International Conference on Machine Learning and Cybernetics*, volume 2, pages 878–882, jul 2008. doi: 10.1109/ICMLC.2008.4620528.

[31] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. Counterfactual Multi-Agent Policy Gradients. *ArXiv e-prints*, May 2017.

[32] Middleware-based LabMap. URL `https://cbb.de/en/products/middleware-based-labmap/`.